Débuter en programmation

2ème ÉDITION

LE GUIDE COMPLET

Avec Visual Basic® 2008 Express Edition

« Réalisez vos propres applications Windows et vos sites Web »

Micro Application

Frédéryk Blot
Yann Lautrédou

2ème Édition - Juillet 2009

Auteurs Frédéryk BLOT, Yann LAUTREDOU

ISBN : 978-2-300-023873

MICRO APPLICATION
20-22, rue des Petits-Hôtels
75010 PARIS
Tél. : 01 53 34 20 20
Fax : 01 53 34 20 00
http://www.microapp.com

Support technique :
Également disponible sur
www.microapp.com

Retrouvez des informations sur cet ouvrage !

Rendez-vous sur le site Internet de Micro Application **www.microapp.com**. Dans le module de recherche, sur la page d'accueil du site, entrez la référence à 4 chiffres indiquée sur le présent livre.
Vous accédez directement à sa fiche produit.

→ RECHERCHE
• PAR MOTS CLÉS
2387
OK

Avant-propos

Destinée aussi bien aux débutants qu'aux utilisateurs initiés, la collection *Guide Complet* repose sur une méthode essentiellement pratique. Les explications, données dans un langage clair et précis, s'appuient sur de courts exemples. En fin de chaque chapitre, découvrez, en fonction du sujet, des exercices, une check-list ou une série de FAQ pour répondre à vos questions.

Vous trouverez dans cette collection les principaux thèmes de l'univers informatique : matériel, bureautique, programmation, nouvelles technologies...

Conventions typographiques

Afin de faciliter la compréhension des techniques décrites, nous avons adopté les conventions typographiques suivantes :

- **gras** : menu, commande, boîte de dialogue, bouton, onglet.
- *italique* : zone de texte, liste déroulante, case à cocher, bouton radio.
- `Police bâton` : Instruction, listing, adresse internet, texte à saisir.
- ✂ : indique un retour à la ligne volontaire dû aux contraintes de la mise en page.



Il s'agit d'informations supplémentaires relatives au sujet traité.



Met l'accent sur un point important, souvent d'ordre technique qu'il ne faut négliger à aucun prix.



Propose conseils et trucs pratiques.



Donne en quelques lignes la définition d'un terme technique ou d'une abréviation.

Introduction

Qu'est ce que la programmation ?

Bien souvent, la première idée que l'on a de la programmation ressemble à celle que l'on se fait d'un cours de mathématiques : une horde de notions techniques et théoriques dont la compréhension est réservée à une élite d'informaticiens. C'est totalement faux. En adoptant une définition un tant soit peu élargie, nous pouvons dire que nous sommes tous un peu programmeurs. Pourquoi ? Simplement parce que nous avons tous autour de nous des appareils électroménagers à qui nous pouvons donner des ordres et qui fournissent en réponse le résultat voulu. Peut-être avez-vous déjà demandé à votre magnétoscope d'enregistrer une émission alors que vous étiez absent ?

La programmation n'est rien d'autre que demander à un ordinateur de faire ceci, puis cela et d'afficher le résultat, le tout dans une langue qu'il comprend. Bien sûr, le clavier d'un PC a plus de touches que la télécommande d'un magnétoscope. Mais cela ne signifie pas que c'est plus compliqué. Au contraire, un ordinateur parle un langage qui est presque proche du nôtre. L'ensemble du monde de la programmation se résume alors à apprendre à parler une langue que comprend la machine et avec laquelle vous allez pouvoir vous entendre. Comme nous allons le voir au long de ce livre, cette tâche est bien plus aisée et amusante qu'il n'y paraît.

Une fois que vous parlerez un langage que votre ordinateur comprend, vous pourrez alors lui donner une liste d'instructions qui vont constituer un programme autrement appelé "logiciel".

Pourquoi programmer ?

Vous vous demandez peut-être quel est l'intérêt de savoir programmer puisqu'un ordinateur est livré avec un lot de logiciels et pourquoi prendre du temps à essayer de développer vous-même des programmes…

Si vous avez déjà été bloqué par les logiciels fournis avec votre PC, si vous n'avez pas réussi, avec votre logiciel photo, à mettre des oreilles de chat sur le portrait de votre patron, ou encore si vous souhaitez ajouter

d'autres niveaux à votre jeu préféré, cela justifie que vous vous intéressiez à la programmation. Comme énoncé en introduction, programmer, c'est donner à un ordinateur une suite d'instructions pour arriver à un résultat. Ainsi, la première raison de s'intéresser à la programmation, c'est de pouvoir dire à la machine ce qu'elle doit faire.

Peut-être cherchez-vous simplement un nouveau passe-temps, voire un nouveau métier ? En apprenant à programmer, vous ouvrez les portes d'un monde infini de possibilités. En effet, dès lors que vous savez programmer, tout devient possible : qu'il s'agisse de jeux, de faire les comptes, de retoucher des photos, d'organiser la navigation Internet… La liste est sans fin.

Choisir un langage de programmation

Programmer n'est rien d'autre que donner des ordres à un ordinateur dans un langage qu'il comprend. Mais qu'est-ce qu'un langage de programmation ? Comme tout langage humain, il est composé de mots à mettre dans le bon ordre, avec la bonne ponctuation, et selon des règles de syntaxe à respecter pour qu'ils soient compris par l'ordinateur. Le nombre de mots se limite à une cinquantaine et, pour ce qui est des règles, on en compte une dizaine, ce qui rend l'apprentissage beaucoup plus facile, comparativement à n'importe quelle langue vivante.

Les langages de programmation sont apparus progressivement, pour répondre aux besoins informatiques. Avec les premiers ordinateurs, la programmation consistait à entrer une suite de 0 et de 1, seuls éléments compréhensibles par la machine et qui correspondaient au passage du courant pour le 1 et à une absence de passage pour le 0. Ce langage binaire obligeait le développeur à parler le même langage que le matériel composant l'ordinateur, et dès que les programmes faisaient plus d'une centaine de lignes, leur relecture était un vrai calvaire.

Ainsi, est apparu très vite le besoin d'un langage intermédiaire, compréhensible à la fois par l'homme et par la machine. Le premier de ces langages a été l'assembleur. Encore proche de la machine, sa syntaxe ressemblait à ceci :

```
ADD(1,2) ;
SUBSTR(5,4) ;
```

En lisant ces deux lignes on peut avoir une vague idée de ce que fait le programme, mais le côté naturel n'est pas encore présent. Apparaissent

alors dans les années 70 et 80 des langages comme le C, plus proches de l'homme, comme le montrent les lignes suivantes :

```
If(x>10)
Printf("x est supérieur à 10") ;
Else
Printf("x est inférieur à 10") ;
```

Ici, avec quelques notions d'anglais, le langage apparaît presque compréhensible dès lors que l'on sait qu'en langage C, `printf` permet un affichage à l'écran.

En 1983, la micro-informatique n'est en aucun cas comparable avec ce qu'elle est aujourd'hui. IBM lance tout juste son ordinateur personnel, avec un rêve : un ordinateur par foyer.

À la fin des années 80 apparaissent des langages qui permettent à un utilisateur de réaliser de manière simple des applications complètes et attrayantes. Se développe alors la programmation grand public, hors du contexte totalement universitaire.

Dix ans plus tard, sont développés des langages comme Java de Sun et C# de Microsoft. Leur intérêt est de permettre au développeur de ne plus avoir à connaître la machine. Ils facilitent grandement la programmation, puisque le programmeur peut maintenant se concentrer sur l'apprentissage du langage sans avoir forcément de connaissances sur l'ordinateur auquel le programme est destiné.

Dans ce livre, nous utiliserons Visual Basic .NET de Microsoft. Ce langage est plus accessible que le C, traditionnellement utilisé dans les cursus universitaires.

Bien préparer la machine

Maintenant que vous savez ce qu'est un langage et à quoi sert la programmation, la dernière étape est de savoir comment parler à un ordinateur pour lui donner des ordres. Pour cela, deux logiciels sont nécessaires.

L'éditeur

Dans un premier temps, il va falloir donner vos ordres à l'ordinateur. Pour cela, vous utiliserez un éditeur de texte. Dans ce livre, nous avons choisi Visual Basic Express 2008, qui regroupe un éditeur et tous les outils dont vous aurez besoin. Il est entièrement gratuit, et vous pourrez le trouver à l'adresse **http://msdn.microsoft.com/fr-fr/express**.

L'éditeur permet, comme tout traitement de texte, de saisir du texte. Comme nous le verrons plus tard, il ne diffère aucunement d'un éditeur de texte classique.

Une fois le téléchargement du programme d'installation de Visual Basic Express 2008 terminé, vous disposez du fichier *vbsetup.exe*. Double-cliquez dessus pour lancer l'installation du programme.



Figure 1 : *Bienvenue dans l'installation*

Une fois sur cet écran, cliquez sur **Suivant**.

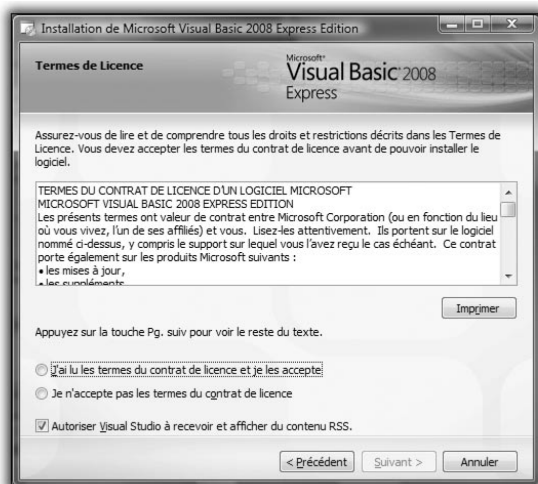


Figure 2 : *Contrat d'utilisation*

Voici le contrat de licence utilisateur. Il est conseillé de le lire attentivement. Cochez la case *J'accepte les termes du contrat de licence*, puis cliquez sur **Suivant**.



Figure 3 : *Vérification de l'espace*

Vérifiez que vous avez suffisamment d'espace disque disponible, puis cliquez sur **Installer**.

Le programme d'installation télécharge les composants nécessaires et les installe sur votre ordinateur. L'opération peut être longue.



Figure 4 : *Fin de l'installation*

Le compilateur

La programmation consiste en un ensemble d'ordres passés à l'ordinateur via un éditeur de texte.

Mais est-ce que la machine comprend directement les instructions qui lui sont passées via cet éditeur ? En fait, non. En début de chapitre, nous avons dit qu'un langage de programmation est compris à la fois par l'homme et par la machine. Ce n'est pas tout à fait vrai. En fait, il n'est pas compris par la machine, mais par un programme spécifique appelé "compilateur". C'est le compilateur qui transforme ce que vous saisissez en un langage utilisé par la machine.

Le fonctionnement d'un compilateur est simple. Une fois que vous avez terminé d'entrer vos instructions dans un fichier, qui est appelé "source", vous appelez le compilateur (nous expliquerons comment ultérieurement). Celui-ci vérifie la syntaxe du fichier, le traduit en

langage machine et place le résultat dans un fichier de type exécutable. On dit alors que le fichier source a été compilé. Il suffira ensuite de double-cliquer sur ce fichier pour que les instructions que vous avez données soient exécutées, les unes à la suite des autres.

Visual Basic Express s'accompagne d'un compilateur qui fonctionne de manière transparente vis-à-vis de l'éditeur. Nul besoin de connaître une longue suite de commandes s'apparentant à de la magie : une touche suffit pour compiler et générer un programme de manière à le rendre exécutable. Maintenant que vous savez ce que sont les compilateurs et les éditeurs de texte, et que votre machine est prête, passons à un cas pratique…



Versions de Visual Studio Express

Les versions 2005 et 2008 sont presque totalement compatibles entre elles. Si vous aviez déjà installé une version 2005 de Visual Studio Express, vous pouvez très bien l'utiliser en lieu et place de la version 2008. Le code présenté dans cet ouvrage est compatible avec la version 2005 de Visual Studio Express.

Chapitre 1

Premiers pas

1.1. Créer un premier projet

Pour commencer, lancez Visual Basic Express. Double-cliquez sur le raccourci placé sur votre Bureau ou allez dans le menu **Démarrer/Tous les programmes** et cliquez sur **Visual Basic Express 2008**.

Vous voici devant la page d'accueil de Visual Basic Express 2008.

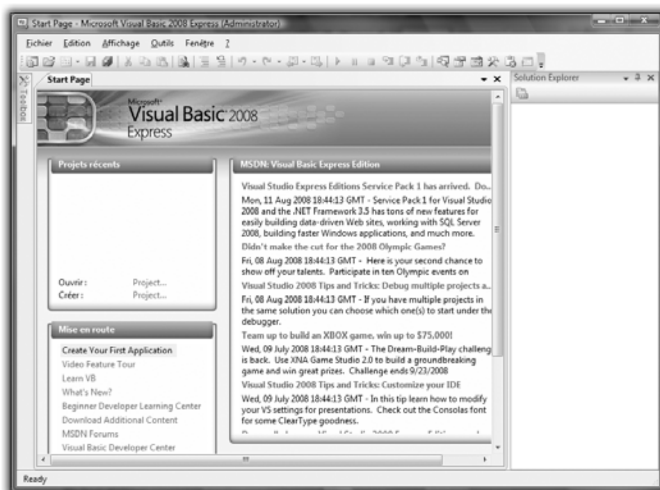


Figure 1.1 : Page d'accueil

Dans le menu **Fichier**, cliquez sur **Nouveau Projet**. Vous arrivez alors au menu de choix suivant :

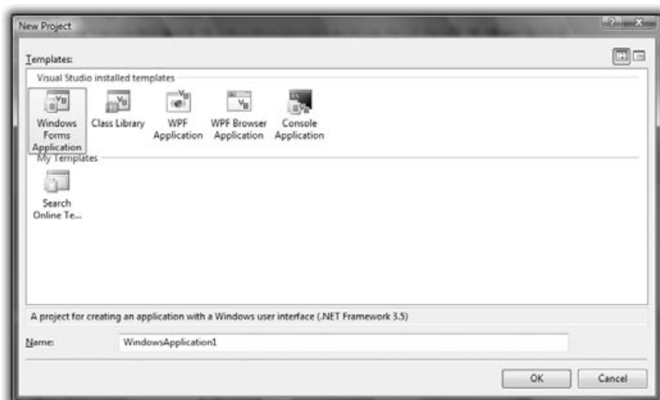


Figure 1.2 : Choix d'application

Vérifiez que l'entrée *Application Windows* est sélectionnée, saisissez un nom pour votre projet en lieu et place de *WindowsApplication1*, par exemple `MonPremierProjet`, et cliquez sur OK.

Vous arrivez à la troisième étape du processus de création de projet. L'écran de Visual Basic ressemble à ceci :

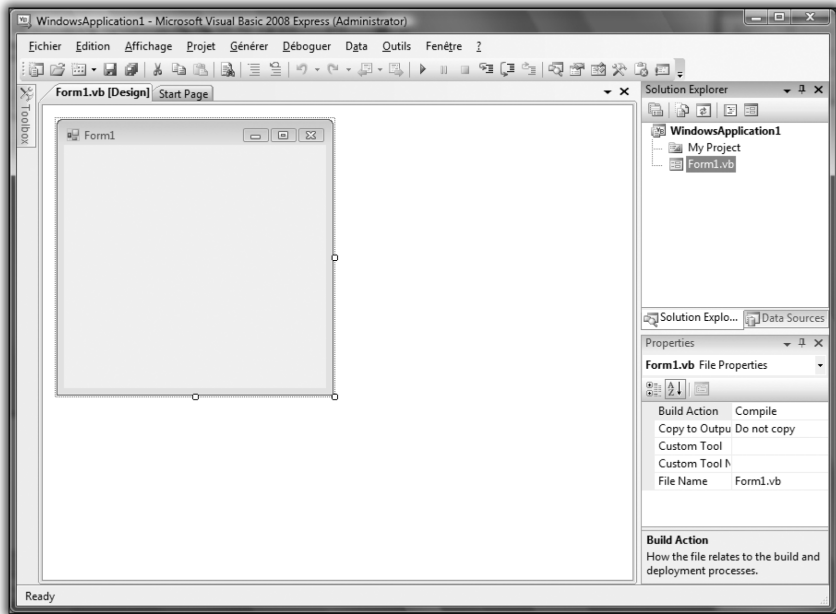


Figure 1.3 : *Prêt à développer !*

1.2. Écrire un programme

Observez ce qui se trouve dans la zone de travail. En haut à droite figure une arborescence contenant deux entrées :

- *MyProject*, qui va regrouper les propriétés relatives au projet en cours. Nous détaillerons son fonctionnement plus tard.
- *Form1.vb*, qui est la fenêtre principale de l'application.

Sous cette arborescence, appelée "Explorateur de solutions", se trouve la fenêtre **Propriétés**.

Repérez le carré gris à gauche de l’espace de travail. Cette zone s’appelle le designer et le carré gris est un formulaire Windows. Une fois que vous aurez fini ce chapitre, il ressemblera à une fenêtre Windows classique.

Cliquez sur le formulaire (le carré gris) et observez la fenêtre des propriétés (en bas à droite de l’espace de travail). Elle se remplit d’une liste de propriétés qui vont définir le formulaire.

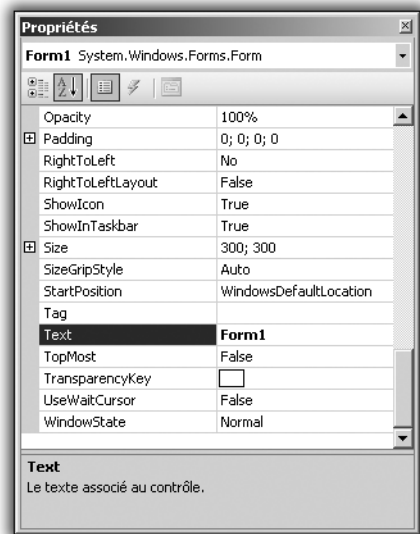


Figure 1.4 :
Aperçu des propriétés

Repérez celle qui s’appelle *Text*. À droite, vous devriez trouver la valeur *Form1*. En lieu et place de *Form1*, saisissez `MaFenêtre`. Appuyez sur la touche ↵ : le titre du formulaire dans le designer change ; vous lisez désormais *MaFenêtre*.

La fenêtre des propriétés permet, entre autres, de modifier le comportement des fenêtres, par exemple leurs titres, leurs couleurs de fond, etc.



*Nous y reviendrons au chapitre **Dialoguer avec un ordinateur**.*

L’espace de travail est maintenant configuré pour le projet en cours et une fenêtre porte le nom souhaité.

Cliquez à gauche sur le bouton nommé **Boîte à outils**. Le menu suivant s'affiche alors :

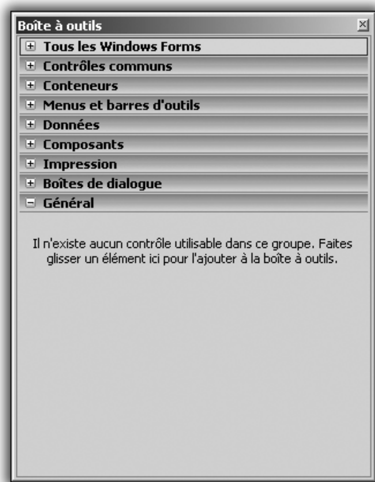


Figure 1.5 :
La boîte à outils

Cette boîte à outils regroupe tous les éléments dont vous avez besoin pour construire une application : des boutons, des zones de saisie de texte, des cadres pour placer des images, etc.

Pour découvrir les possibilités de cette boîte, cliquez sur le menu **Tous les Windows Forms**. Une liste déroulante apparaît. Elle donne le choix entre plusieurs outils, appelés "contrôles". Trouvez celui qui s'appelle *Button*. Effectuez un glisser-déposer vers la fenêtre dans le designer. Votre fenêtre ressemble alors à ceci :



Figure 1.6 :
Première ébauche de la fenêtre

Faites de même avec le contrôle appelé *TextBox*. Vous avez maintenant une fenêtre qui ressemble à celle-ci :



Figure 1.7 : *Fenêtre complète*

1.3. Compiler, exécuter

Votre fenêtre possède un bouton, une boîte permettant de saisir du texte et un nom. À partir de maintenant, vous allez vous intéresser au comportement de ladite fenêtre. Tout d'abord, générez votre programme pour le transformer en exécutable. Pour cela, appuyez sur la touche F5. Votre projet va être généré et lancé. Visual Basic Express 2005 passe alors en mode Test et votre écran ressemble à ceci :

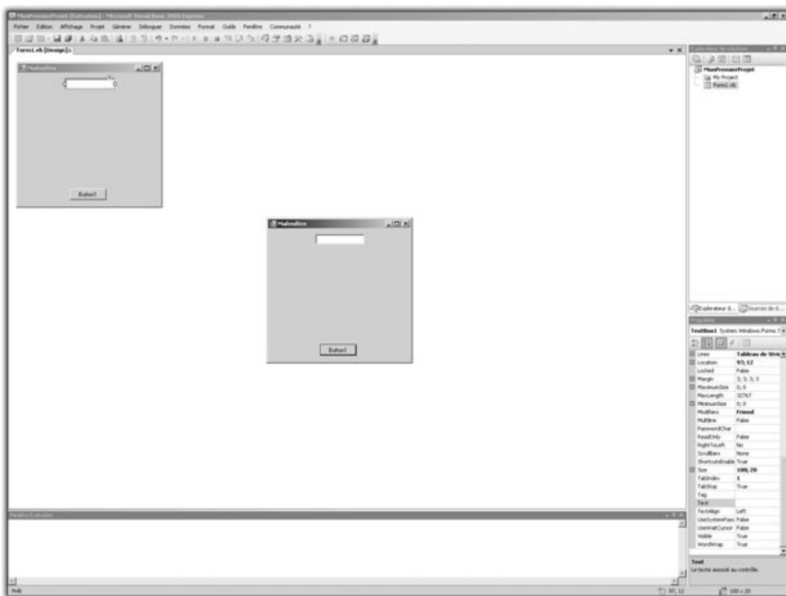


Figure 1.8 : *Votre programme est lancé*

Votre fenêtre est lancée, comme tout autre programme, et dispose de son icône dans la barre des tâches de Windows. Vous pouvez cliquer sur le bouton, entrer du texte dans la *TextBox*, mais rien d'autre. Pour aller plus loin, fermez la fenêtre en cliquant sur la croix. Visual Basic Express 2005 revient automatiquement en mode Développement.

Vous allez entrer vos premières instructions, pour donner un comportement à la fenêtre. Double-cliquez sur le bouton de la fenêtre. Vous vous retrouvez sous un nouvel onglet, face à ce texte :

```
Public Class Form1

    Private Sub Button1_Click(ByVal sender As System.Object,
    ✂ ByVal e As System.EventArgs) Handles Button1.Click

    End Sub
End Class
```

Entre les lignes `Private Sub […]` et `End Sub`, saisissez le texte suivant :

```
MessageBox.Show("Bonjour " + TextBox1.Text)
```

Cela fait, générez votre programme modifié en appuyant sur F5. La fenêtre s'affiche à nouveau. Cette fois, entrez votre prénom dans la *TextBox* puis cliquez sur le bouton. L'affichage qui en résulte est le suivant :



Figure 1.9 :
L'ordinateur vous dit bonjour

Une boîte de dialogue est affichée ; elle contient le message "Bonjour" suivi de votre prénom. Cliquez sur OK et fermez la fenêtre de votre programme. Analysons maintenant ce que vous avez fait.

1.4. Mieux comprendre

Dans ce chapitre, vous avez réalisé un programme complet qui affiche du texte.

Tous vos développements à partir de maintenant se dérouleront de cette manière. Dans un premier temps, créez un nouveau projet dans Visual

Basic. C'est le point de départ indispensable pour obtenir un espace de travail fonctionnel.

Dans cet espace de travail, utiliserez le designer pour placer sur la fenêtre tout ce dont vous aurez besoin pour le programme. Dans ce premier exemple, vous avez placé une *TextBox* et un bouton.

Une fois que la fenêtre contient tous les contrôles nécessaires, définissez le comportement de l'application avec l'éditeur de code de Visual Basic Express. C'est le moment d'entrer toutes les instructions que doit exécuter l'ordinateur. Dans cet exemple, les seules instructions nécessaires sont celles qui sont exécutées lors d'un clic sur le bouton de la fenêtre principale. Comme à chaque clic une seule instruction est exécutée, vous avez saisi une seule ligne dans l'éditeur de code, à savoir :

```
MessageBox.Show("Bonjour " + TextBox1.Text)
```

Cette ligne permet d'afficher ce que vous saisissez dans la *TextBox* après le texte "Bonjour". Mais pourquoi à ce moment-là ? Tout simplement parce que vous avez placé ce texte après avoir cliqué sur le bouton dans le designer de Visual Basic. Par ce clic, l'éditeur a compris que vous vouliez modifier le comportement de l'application et, plus précisément, ce qu'il se passe lorsque vous cliquez sur le bouton de la fenêtre. En fait, chaque contrôle dispose d'événements, qui représentent un clic sur un bouton, un changement de texte dans une *TextBox*, etc. Lorsque vous déclenchez un événement dans le designer, comme le clic sur le bouton dans cet exemple, vous êtes automatiquement redirigé vers la partie du code qui permet de modifier le comportement de l'application quand l'événement est lancé. Dans cet exemple, cela correspond au code :

```
Private Sub Button1_Click(ByVal sender As System
✂ .Object, ByVal e As System.EventArgs) Handles
✂ Button1.Click

End Sub
```

Tout ce qui se trouve entre `Private Sub` et `End Sub` représente le code que le programme doit exécuter lors de l'événement en question.

Étudions maintenant ce qu'il se passe lorsque vous lancez le programme, c'est-à-dire lorsque vous appuyez sur la touche F5.

Lors de la génération du projet, le compilateur se charge de lire, ligne par ligne, ce que vous avez écrit. Ensuite, il traduit le tout en langage

machine. Le résultat est un fichier exécutable qui, par défaut, est placé dans le répertoire *Mes documents\Visual Studio 2005\Projects\MonPremierProjet\MonPremierProjet\bin\Debug*. Ce fichier porte le nom du projet et l'extension *.exe*. Dans ce cas, il s'agit du fichier *MonPremierProjet.exe*. Une fois qu'il est généré, vous pouvez le copier où bon vous semble et l'exécuter d'un double clic. C'est de cette manière que vous pourrez le partager avec vos amis, le distribuer sur Internet ou le sauvegarder sur un CD.

Quand vous double-cliquez sur ce programme, vos instructions sont exécutées. Les premières d'entre elles ne sont pas les vôtres puisque vous n'avez précisé de comportement que pour le clic sur le bouton. En fait, le programme commence toujours par les instructions de dessin de la fenêtre. Elles permettent l'affichage d'une fenêtre avec les boutons **Fermer**, **Réduire** et **Agrandir**. Cela fait, le programme dessine l'intérieur de la fenêtre. Dans cet exemple, il place un bouton et une *TextBox*.

Le programme est maintenant chargé. Il attend une action de votre part pour réagir, ici un clic sur le bouton. Le programme enregistre le fait que vous cliquez sur le bouton, regarde sa liste d'instructions et voit qu'en cas de clic, il faut afficher une boîte de dialogue contenant le message "Bonjour xxx", où xxx est le texte saisi dans la *TextBox*.

Une fois la boîte de dialogue affichée, le programme attend que vous cliquiez sur le bouton OK pour continuer et repasse à la première étape : il attend d'autres instructions. Vous pouvez par exemple saisir un autre nom dans la *TextBox* ou encore quitter le programme en cliquant sur la petite croix en haut à droite de la fenêtre.

Dans ce chapitre, vous avez vu les principes fondamentaux de création de projet, d'édition de code et de lancement de programme. Voyons maintenant comment enrichir le contenu des projets par un code plus complet…

Chapitre 2

Dynamiser un programme

Le premier programme que vous avez créé dans le chapitre précédent est complet, mais il ne réalise rien de réellement important ou d'utile. Vous allez donc à présent donner à ce programme une réelle dynamique, grâce à l'utilisation de variables.

Qu'est-ce qu'une variable ? Il s'agit tout simplement d'un petit espace que vous réservez à l'intérieur de la mémoire de votre ordinateur pour y stocker une valeur d'un type donné. Bien évidemment, il n'est pas utile de connaître le fonctionnement interne de votre machine pour utiliser des variables.

En fait, la mémoire de la machine fonctionne exactement comme la vôtre. Par exemple, pour mémoriser le numéro de téléphone de Sylvie, votre cerveau alloue un petit espace dans lequel il stocke 06.12.34.56.78, et l'associe à Sylvie en se rappelant qu'il s'agit d'un numéro de téléphone.

Pour stocker une variable dans la mémoire d'un ordinateur, il faut raisonner de la même manière, c'est-à-dire qu'il faut lui donner un nom unique et indiquer son type (nombre, chaîne de caractères…).

2.1. Déclarer une variable

Lancez Visual Basic Express et créez un nouveau projet. Appelez-le `ProjetVariables`. Une fois dans le designer, double-cliquez sur la fenêtre pour lancer l'éditeur de code. Une fois dans le code, ajoutez ces deux lignes à l'endroit où est placé le curseur :

```
Dim num As Integer
  num = 100
```

Avec ses deux lignes, vous précisez à l'ordinateur qu'il doit stocker une variable de type nombre entier. Vous lui donnez la valeur 100 et vous l'appelez `num`. Examinons le contenu de ce code :

- `Dim` : ce mot-clé du langage permet de spécifier que vous déclarez une variable.
- `num` : c'est le nom de la variable.
- `As` : ce mot-clé permet de spécifier le type de la variable utilisée.
- `Integer` : il s'agit du type nombre entier.
- `num = 100` : cette opération affecte à la variable `num` la valeur 100.

Vous avez à présent dans le programme une variable nommée `num` qui contient la valeur 100. C'est aussi simple que cela. Voyons maintenant comment l'utiliser.

2.2. Utiliser une variable

Pour utiliser une variable dans un programme, il suffit de placer son nom à l'endroit où vous désirez utiliser sa valeur. Pour l'exemple du numéro de téléphone, il suffirait d'écrire "Sylvie" et l'ordinateur saurait qu'il faut remplacer ce libellé par 06.12.34.56.78. Ainsi, l'exemple suivant affiche la valeur de la variable `num`. Replacez-vous dans l'éditeur de code et ajoutez ces deux lignes :

```
num = num+100
MessageBox.Show(num.ToString)
```

Appuyez maintenant sur F5. Le programme se lance, avec une boîte de dialogue qui donne la valeur de `num`, à savoir 200.

Ces deux lignes permettent d'afficher la nouvelle valeur de `num`, à laquelle vous avez ajouté 100. L'opération `ToString` permet de transformer un type nombre en chaîne de caractères en vue de son affichage. Il existe un grand nombre d'opérations possible utilisant les variables. Il est maintenant temps de se familiariser avec les différents types de variables.

Parmi ceux-ci existent les booléens, qui permettent de raisonner sous forme de conditions, les nombres entiers et décimaux, qui permettent d'effectuer des calculs, et les caractères et chaînes de caractères, qui permettent de manipuler du texte. Chacun de ces types est incompatible avec les autres et vous ne pourrez donc pas additionner un nombre avec du texte par exemple. Comme vous l'avez vu plus haut, il est possible de convertir un type en un autre à l'aide de méthodes déjà existantes. C'est le cas de `num.ToString`, qui transforme le nombre `num` en chaîne de caractères. Voici le détail de chaque type de variable et de ce qu'il permet.

2.3. Les booléens

Les booléens sont des variables issues de la logique du mathématicien George Boole. Cette logique permet une représentation par état, qui donne le passage du courant comme étant égal à 1 et le non-passage du

courant comme étant égal à 0. Elle est à la base de l'électronique et de l'informatique.

Dans une réflexion de programmation, déclarer une variable booléenne permet de réagir selon certains états. Dans chaque variable booléenne déclarée, on peut stocker un état, par exemple si une porte est ouverte ou non. Un booléen ne peut prendre que deux valeurs : "vrai" ou "faux".

Voici pour la mise en pratique :

1. Ouvrez un nouveau projet avec Visual Basic Express et nommez-le `ProjetBool`. Puis, dans le designer, placez-vous dans l'éditeur de code en double-cliquant sur la fenêtre.
2. À l'endroit du curseur, saisissez le code `Dim monBoo As Boolean`, ce qui a pour effet de déclarer une variable de type booléenne, de nom `monBoo`.
3. Retournez à la ligne et saisissez `monBoo =`. Lorsque vous entrez le caractère =, l'éditeur de texte vous propose d'attribuer une valeur au booléen. Pour cet exemple, choisissez `True`.
4. Ajoutez la ligne `MessageBox.Show(monBoo.ToString)`.
5. Lancez le programme en appuyant sur F5.

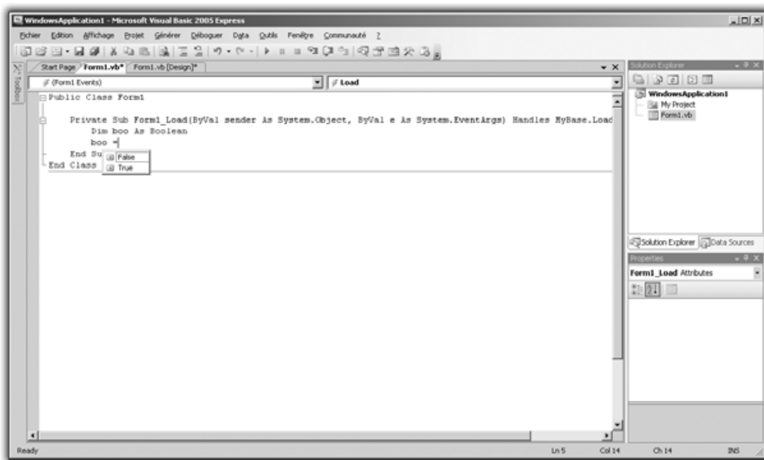


Figure 2.1 : *Affichage du booléen*

L'utilisation de telles variables permet de réaliser des tests dans les applications. Pour votre première application, il aurait par exemple été

pertinent d'utiliser un booléen pour vérifier que l'utilisateur a bien entré son prénom avant d'appuyer sur le bouton OK.



Vous verrez comment faire cela aux chapitres ***Contrôler un programme*** *et* ***Dialoguer avec un ordinateur****.*

2.4. Les nombres

Les nombres entiers

Que vous réalisiez une application de manipulation d'images, de comptabilité, de calendrier ou d'annuaires, vous aurez forcément besoin de manipuler des nombres. Nous avons déjà vu en introduction de ce chapitre que vous pouviez initialiser une variable en tant que type numérique entier déclaré à l'aide du mot-clé `Integer`.

En plus de la possibilité de déclarer des variables de type entier, vous pouvez réaliser toute une série d'opérations sur ces nombres : par exemple l'addition, la soustraction, la multiplication, la division, la division entière, etc.

Pour initialiser une variable de type nombre entier, utilisez la commande suivante :

```
Dim MaVariable As Integer
MaVariable = 5
```

Cela fait, la variable nommée `MaVariable` possède la valeur 5. Initialisez maintenant une nouvelle variable de type entier :

```
Dim MaVariable2 As Integer
MaVariable2 = 3
```

Pour tester les opérations possibles entre ces deux variables, vous allez initialiser une troisième variable de type entier, qui permettra de stocker le résultat d'une opération :

```
Dim MonResultat As Integer
MonResultat = MaVariable + MaVariable2
```

Ici les deux valeurs sont additionnées. Libre à vous de changer le caractère `+` pour `*`, qui réalise la multiplication des deux valeurs, ou encore `/`, qui réalise une division.

Affichez maintenant le résultat :

```
MessageBox.Show(MonResultat.ToString)
```

Pour tester cet exemple, créez un nouveau projet dans Visual Studio Express, puis double-cliquez sur la fenêtre dans le designer. Une fois dans l'éditeur de code, saisissez les instructions précédentes. Lancez le programme avec F5.



Figure 2.2 :
Résultat de l'addition de deux nombres

Ajouter des virgules

Nous avons vu le cas des nombres entiers. Mais que se passe-t-il si vous stockez une variable contenant le résultat de la division de deux entiers dans un autre nombre entier ? Si la division tombe juste, tout va bien. Par contre, si le résultat est un nombre à virgule, plus rien ne marche !

En effet, l'ordinateur n'étant doué d'aucune intelligence, il ne comprend pas comment faire entrer un nombre d'un certain type dans une variable qui n'est pas de ce type. Il faut donc recourir à un autre type de variable : `Decimal`.

Voici un exemple de programme que vous pouvez reprendre par exemple pour calculer une moyenne :

```
Dim x As Integer
x = 10
Dim y As Integer
y = 14
Dim z As Integer
z = 8
Dim res As Decimal
res = ((10+14+8)/3)
MessageBox.Show(res.ToString)
```



Figure 2.3 :
Résultat de la moyenne



Déclarer une variable tout en l'initialisant

Lorsque vous déclarez des variables, il n'est pas utile de préciser le type si vous faites tout de suite l'initialisation. Lorsque le programme sera compilé, le compilateur va automatiquement reconnaître le type de variable. Par exemple, `Dim x = 42` est équivalent à `Dim x As Integer` suivi de `x = 42`.

2.5. Les jeux de lettres

Les caractères

Après les chiffres, voyons les variables de type caractère et chaîne de caractères.

Un caractère peut être un chiffre, une lettre, l'apostrophe, la parenthèse ou encore le retour chariot et même l'espacement.

Pour déclarer une variable de type caractère, utilisez la syntaxe suivante :

```
Dim c As Char
c = 'f'
```

Notez que les caractères sont initialisés à l'aide d'une valeur entre apostrophes. C'est le cas du `'f'` dans cet exemple.

Les variables de type caractère peuvent être utiles dans le cadre de travaux sur les mots, par exemple si vous souhaitez vérifier que la première lettre d'un mot est une majuscule. Cela est possible à l'aide d'outils proposés dans Visual Basic Express.



Tout cela sera détaillé au chapitre ***Contrôler un programme.***

Les chaînes

Plusieurs caractères mis les uns après les autres forment une chaîne de caractères. Ces chaînes sont très utilisées en programmation dans la mesure où il faut souvent demander à l'utilisateur d'un programme de

saisir des données. S'il faut traiter ces données sous forme de texte, les chaînes de caractères entrent en jeu.

Pour déclarer une chaîne de caractères, procédez de la manière suivante :

```
Dim S As String
S = "Bonjour"
```



Subtilités

Notez que les chaînes de caractères sont initialisées entre guillemets ("), et les caractères entre apostrophes (').

Dans la mesure où l'on travaille souvent sur des chaînes de caractères en programmation, un bon nombre de traitements de base ont déjà été mis au point. Ainsi, pour concaténer deux chaînes de caractères, il suffit de faire comme si vous les additionniez :

```
Dim S1 As String
S1 = "Bonjour"
Dim S2 As String
S2 = " tout le monde"
MessageBox.Show(S1+S2)
```



Figure 2.4 :
Concaténation de deux chaînes

De nombreuses autres méthodes sont disponibles pour le travail sur des chaînes. Par exemple, vous pouvez connaître le nombre de caractères d'une chaîne grâce à l'attribut `length`. Pour l'utiliser, il suffit d'ajouter `.length` à la fin du nom d'une variable de type chaîne :

```
Dim S As String
S = "Bonjour"
MessageBox.Show(S.Length.ToString)
```



Figure 2.5 :
Longueur d'une chaîne

Pour convertir une chaîne de caractères en minuscules ou en majuscules, utilisez les fonctions `ToLower` et `ToUpper`, en ajoutant `.ToUpper` à la fin du nom de votre variable chaîne de caractères. Le code suivant stocke une chaîne convertie en minuscules dans une autre variable de type chaîne de caractères et l'affiche :

```
Dim S As String
S = "BONJOUR"
Dim S2 As String
S2 = S.ToLower
MessageBox.Show(S2)
```



Figure 2.6 : *Chaîne en minuscules*

"BONJOUR" est affiché via `S2` sous forme de "bonjour".

Cas pratique : crypter des messages

Après ces exemples d'utilisation de chaînes, vous allez réaliser une application qui cryptera des messages. Grâce à programme, vous pourrez saisir un message qui ne sera lisible que par quelqu'un disposant de votre application de décryptage.

Dans un premier temps, créez un nouveau projet dans Visual Basic Express. Appelez-le `ProjCryptage` par exemple.

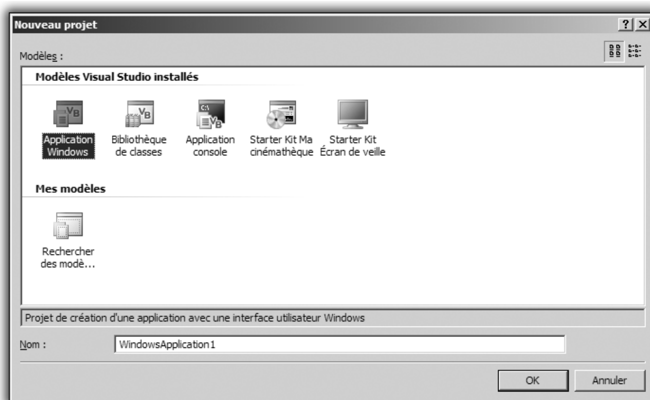


Figure 2.7 : *Création du projet*

Dans le designer, ajoutez à la fenêtre en cours deux *TextBox* et un bouton de sorte que votre fenêtre ressemble à ceci :

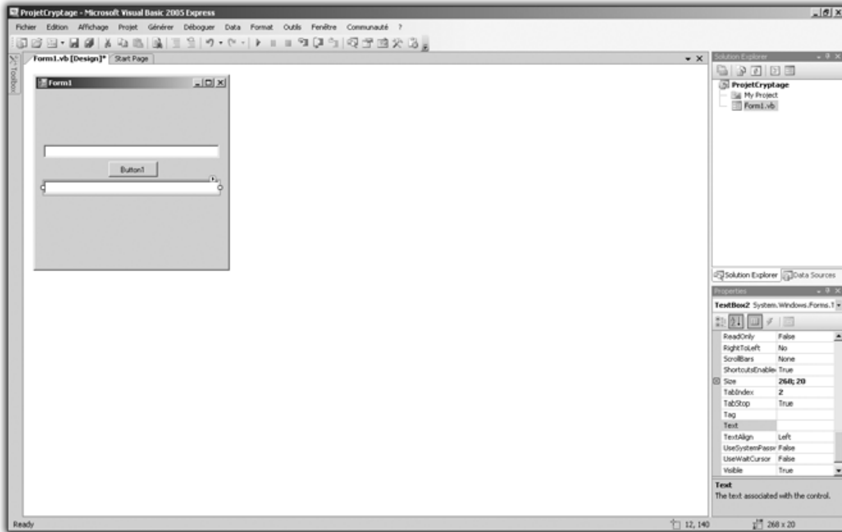


Figure 2.8 : *Design de la fenêtre*

La première *TextBox* est réservée à la saisie du message, la seconde accueillira le message une fois crypté.

Maintenant que le programme est dessiné, il ne reste qu'à écrire le processus de cryptage lorsque l'on clique sur le bouton de l'application.

Pour cela, double-cliquez sur le bouton. Dans l'éditeur de code, à l'endroit où se trouve le curseur, ajoutez les lignes suivantes :

```
Dim s As String
s = TextBox1.Text
s = s.Replace("d", "3")
s = s.Replace("e", "7")
s = s.Replace("g", "1")
s = s.Replace("c", "8")
s = s.Replace("q", "2")
s = s.Replace("h", "g")
s = s.Replace("n", "d")
s = s.Replace("t", "5")
s = s.Replace("j", "6")
s = s.Replace("a", "4")
TextBox2.Text = s
```

Ces lignes d'instruction ont pour effet de copier, dans un premier temps, le contenu de la première *TextBox* dans une variable temporaire. Ensuite, tout le traitement va être fait sur cette variable temporaire. Vous utiliserez la méthode `Replace`, disponible pour toutes les chaînes. Elle permet d'échanger deux caractères dans une chaîne. Une première instruction remplacera toutes les lettres "d" de la phrase par des caractères "3", une seconde tous les "e" par des "7", etc. Toutes ces modifications seront effectuées au moment où vous cliquerez sur le bouton. Une fois le traitement fini, la phrase cryptée apparaîtra dans la deuxième *TextBox*, située sous le bouton.

Ainsi, une phrase telle que "J'ai lancé la cafetière" apparaîtra sous la forme : `"6'4i l4d8é l4 84f75i7r7"`. Bien malin qui pourra retrouver le message d'origine !

Voyons à présent comment décrypter le message.

Rien de plus simple : il suffit de refaire la même application, mais en inversant l'ordre des instructions. Répétez les deux premières étapes de ce projet, puis copiez les instructions suivantes après avoir double-cliqué sur le bouton. Appelez ce projet `ProjDecrypt`.

```
Dim s As String
      s = TextBox1.Text
      s = s.Replace("d", "n")
      s = s.Replace("3", "d")
      s = s.Replace("g", "h")
      s = s.Replace("7", "e")
      s = s.Replace("1", "g")
      s = s.Replace("8", "c")
      s = s.Replace("2", "q")
      s = s.Replace("5", "t")
      s = s.Replace("6", "j")
      s = s.Replace("4", "a")
 TextBox2.Text = s
```

Testez le projet en le lançant avec F5 : la chaîne est bien décryptée !

2.6. Convertir les types

Tout au long de chaque développement, vous devrez transformer des chiffres en chaînes de caractères, et inversement. Même des booléens peuvent être transformés en chaînes ou en types numériques.

Passer d'un entier à une chaîne de caractères

Pour passer d'un entier à une chaîne, il suffit d'utiliser la méthode `ToString`. Elle permet de transformer un entier en un type chaîne utilisable directement.

Voici un exemple d'utilisation de `ToString` :

```
Dim x As Integer
x = 42
MessageBox.Show(x.ToString)
```

Si vous essayez de lancer le programme sans ajouter l'instruction `.ToString` dans `MessageBox.Show(x)`, une erreur surviendra et le programme ne se lancera pas.



Conversion de décimaux

Ce type de conversion marche aussi avec les nombres décimaux.

Transformer une chaîne de caractères en nombre entier

Pour passer d'une chaîne à un nombre, il faut utiliser la méthode `Int32.Parse(votrechaine)`. Elle renvoie un entier qu'il convient de stocker dans une variable appropriée. Voici un exemple d'utilisation :

```
Dim x As Integer
Dim S As String
S = "123456"
x = Int32.Parse(S)
```

Dans ce cas, `x` sera égal au nombre 123 456. Il est nécessaire de réaliser cette conversion si vous désirez effectuer des opérations sur une saisie de l'utilisateur par exemple. Rappelez-vous que l'opérateur + ne veut pas dire la même chose selon que l'on manipule des entiers ou des chaînes de caractères. Examinons le code suivant :

```
Dim s As String
s = "123"
Dim s2 As String
s2 = "456"
MessageBox.Show(s+s2)
```

Il n'affichera pas 579 mais 123 456. Attention donc au type des variables. Pour additionner deux entiers d'abord représentés sous forme de chaînes, vous devez les convertir puis les stocker à part, comme le montre l'exemple suivant :

```
Dim s As String
s = "123"
Dim s2 As String
s2 = "456"
Dim x As Integer
x = Int32.Parse(s)
Dim Y As Integer
Y = Int32.Parse(s2)
Dim Addi As Integer
Addi = x+y
```

Chapitre 3

Des variables plus complexes

3.1. Les énumérations

Définition

Ce type correspond à une constante entière, à laquelle vous donnez un nom de manière à la rendre plus explicite. Pour une couleur par exemple, la variable sera une énumération de type `couleur`, dont les valeurs seront par exemple `Vert`, `Jaune`, ou `Rouge`, etc.

Cependant, derrière ces noms se cache ni plus ni moins qu'une valeur entière. Les énumérations ne sont vraiment qu'un moyen pour rendre l'écriture plus agréable et plus compréhensive, quand il y a un nombre fini de valeurs possibles.

Déclarer une énumération

Voici comment déclarer une énumération :

```
Public Enum VetementHaut
  Tshirt
  Chemise
  Debardeur
End Enum
```

Lorsque vous ne spécifiez aucune valeur dans l'énumération, la valeur entière correspondante commence à 0, et va en augmentant. Dans ce cas, `Vert` vaut donc 0, `Jaune` vaut 1, et `Rouge` vaut 2. Cependant, vous pouvez spécifier des valeurs pour qu'elles soient plus pertinentes. Par exemple :

```
Public Enum VetementBas
  Pantalon = 1
  Jeans = 2
  Bermuda = 3
End Enum
```

Utiliser des énumérations

Les énumérations s'utilisent ensuite comme des variables normales. Après avoir déclaré le type comme décrit précédemment, il faut déclarer une variable de ce type comme ceci :

```
Dim haut As VetementHaut
```

Vous pouvez ensuite affecter une valeur à cette variable. Pour donner à la variable une valeur de l'énumération, il faut indiquer le type et la valeur en question, séparés par un point :

```
haut = VetementHaut.Chemise
```

Vous pouvez d'ailleurs initialiser la variable directement pendant la déclaration :

```
Dim bas As VetementBas = VetementBas.Pantalon
```

3.2. Les enregistrements

Définition

Un enregistrement (on peut également parler de "structure") est une variable contenant un ensemble de données de types différents, simples ou structurés. Ils permettent de regrouper un certain nombre d'informations liées à la même chose, ou encore de représenter une information formée de plusieurs composantes, par exemple des coordonnées géographiques.

C'est en fait un groupement de variables qui ne sont alors plus indépendantes car rassemblées autour d'un même élément englobant.

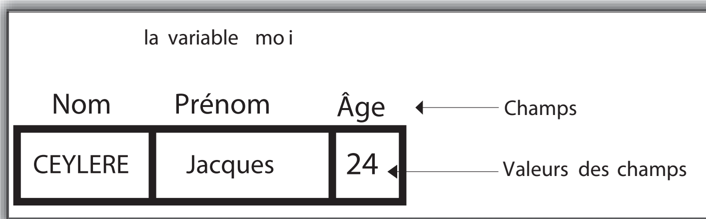


Figure 3.1 : *Représentation d'un enregistrement*

Déclarer un enregistrement

Pour cela, utilisez le mot-clé `Structure` et, dans le bloc de déclaration, indiquez les différentes composantes de l'enregistrement :

```
Public Structure Personne
  Public String nom
  Public String prenom
```

```
  Public Integer age
End Structure
```

Chaque composante de l'enregistrement est appelée "champ". Ici, il y en a trois : le nom, le prénom, et l'âge, de type chaîne de caractères pour les deux premiers, et entier pour le dernier.

Utilisation des enregistrements

L'utilisation des enregistrements se fait par la lecture et/ou l'écriture de ses différents champs.

Il faut d'abord déclarer la variable et l'instancier :

```
Dim moi As Personne
moi = New Personne()
```

La deuxième ligne instancie la variable, c'est-à-dire qu'elle crée en mémoire l'espace nécessaire à son utilisation. En effet, un enregistrement étant un regroupement de champs éventuellement de types différents et dont le nombre n'est pas directement limité, sa taille n'est pas fixe. L'instanciation sert donc à déterminer l'espace exact nécessaire pour que vous puissiez utiliser cette variable complètement.

Une fois qu'elle est instanciée, tout l'espace est créé, tous les champs sont accessibles par leur nom, précédé du nom de la variable et d'un point. Vous pouvez alors les lire, leur affecter des valeurs, comme n'importe quelle variable :

```
moi.nom = "CEYLERE"
moi.prenom = "Jacques"
moi.age = 24
MessageBox.Show("Je suis " + moi.nom + " " + moi.prenom)
MessageBox.Show("J'ai " + moi.age + " ans.")
```

Un enregistrement est un type structuré dont les champs peuvent être de différents types. C'est pourquoi il n'est pas aisé d'initialiser directement un enregistrement pendant la déclaration. Des mécanismes permettent de le faire lors de l'instanciation (nous y reviendrons). Pour l'instant, on supposer que, pour initialiser un enregistrement, il faut initialiser chacun de ses champs juste après sa création, ce qui peut s'avérer fastidieux si l'enregistrement est conséquent.

3.3. Les tableaux

Définition

Un tableau est une variable contenant un ensemble de données de même type. Ces données peuvent être de types simples (tableaux d'entiers ou de caractères) ou complexes (tableaux d'enregistrements, de chaînes de caractères ou tableaux de tableaux).

Un tableau peut avoir une, deux, trois ou n dimensions. À une dimension, c'est un ensemble de cases "sur une ligne".

12	36	151	7	42	87	51	← Valeurs
0	1	2	3	4	5	6	← Indices

Figure 3.2 : *Tableau à une dimension*

À deux dimensions, on peut le représenter comme ayant des lignes et des colonnes. On l'appelle alors "matrice".

Lignes \ Colonnes	0	1	2	3	
0	51 (0,0)	37 (0,1)	9 (0,2)	12 (0,3)	← Valeurs
1	42 (1,0)	22 (1,1)	81 (1,2)	72 (1,3)	← Indices (2 dimensions)

Figure 3.3 : *Tableau à deux dimensions (matrice)*

À trois dimensions, il aurait des lignes, des colonnes, plus une profondeur.

Déclarer un tableau

Pour déclarer un tableau, vous devez définir un certain nombre de choses. La première est le type des données qui seront à l'intérieur du tableau. En effet, un tableau d'entiers n'est pas pareil (en particulier il n'a pas la même taille) qu'un tableau de caractères. De plus, il faut préciser sa taille, c'est-à-dire le nombre d'éléments qu'il contiendra, et ce pour chacune des dimensions s'il y en a plusieurs. Visual Basic .NET

présente une particularité à ce niveau : il s'agit de définir, non pas la taille, mais l'indice du dernier élément, sachant que les indices commencent à 0.

```
' Déclaration d'un tableau simple de 9 chaînes de caractères
Dim courses(8) As String

' Déclaration d'une matrice de 4 X 3 entiers
Dim maMatrice(3, 2) As Integer
```

Utiliser les tableaux

Une fois qu'un tableau est déclaré, vous pouvez accéder à ses éléments grâce à leurs indices. C'est la position de la donnée dans le tableau. La numérotation commence à 0. En Visual Basic, le dernier élément a l'indice qui a été utilisé lors de la déclaration du tableau.

Il y a autant d'indices qu'il y a de dimensions.

```
courses(0) = "Pâtes"
courses(1) = "Jambon"
Courses(2) = "Beurre"
maMatrice(1)(1) = 42;
MessageBox.Show("En premier je dois prendre : " + courses(0))
MessageBox.Show("Il y a " + maMatrice(1)(1) + " à la case
✂ 2-2")
```

Un tableau est un type structure dont tous les éléments sont du même type. Il est possible d'initialiser le tableau à la déclaration, et ainsi de se retrouver avec un tableau dont les valeurs sont déjà définies. Il suffit de donner la liste des éléments entre accolades, séparés par des virgules :

```
Dim maListe(4) As Integer = { 0, 0, 0, 0, 0 }
```

Cela est pratique lorsque le tableau n'est pas trop grand, mais imaginez ce que cela donnerait avec un tableau de plusieurs centaines d'éléments, voire plusieurs milliers. Dans ce cas, il faut parcourir le tableau avec une boucle, dont nous décrirons le fonctionnement ultérieurement. Voici un avant-goût qui montre comment parcourir un tableau pour l'initialiser. Il faut pour cela utiliser une variable de parcours et donner l'indice de départ (0) et l'indice de fin, que l'on obtient grâce à la méthode `getUpperBound` :

```
Dim tableau(123) As Integer
Dim i As Integer
For i = 0 to tableau.GetUpperBound(0)
    tableau(i) = 23
Next
```

Ici, vous déclarez un tableau de cent vingt-quatre éléments valant tous 23.

3.4. Cas pratique : une bibliothèque multimédia

Vous avez maintenant les éléments de base permettant de construire une application intéressante. En combinant ces éléments, vous pouvez faire des programmes d'un plus grand intérêt. Par exemple, vous allez faire ici une bibliothèque multimédia pour gérer une liste de vidéos.

On peut définir une vidéo par son type d'affichage, à savoir monochrome ou couleur. Pour cela, recourez à une énumération possédant ces deux valeurs :

```
Public Enum Affichage
  Monochrome
  Couleur
End Enum
```

Une vidéo possède un certain nombre d'attributs qui lui sont propres, par exemple son nom, sa longueur, etc. Pour représenter une vidéo, vous pouvez donc utiliser un enregistrement dont les champs correspondent aux différents attributs.

```
Public Structure Video
  Public String nom
  Public Affichage type
  Public Integer longueur
End Structure
```

Vous avez défini les éléments fondamentaux de la bibliothèque. Comme une bibliothèque avec une seule vidéo a peu d'intérêt, vous allez créer un tableau de vidéos pour qu'elle puisse en contenir plusieurs. Il suffit d'utiliser les méthodes vues précédemment :

```
Dim bibliotheque(5) As Video
```

Voila, vous avez décrit tous les types dont vous avez besoin pour gérer la liste de vidéos. Vous pouvez maintenant décrire chacune d'entre elles, puis les mettre dans la bibliothèque, que vous consulterez avant de visionner un film :

```
Dim temps_modernes As Video
temps_modernes = new Video()
temps_modernes.nom = "Les temps modernes"
temps_modernes.type = Monochrome
```

```
temps_modernes.longueur = 89
bibliotheque(0) = temps_modernes
Dim braveheart As Video
braveheart  = new Video()
braveheart.nom = "Braveheart"
braveheart.type = Couleur
braveheart.longueur = 165
bibliotheque(1) = braveheart
```

Vous avez décrit et ajouté deux films à la bibliothèque. Selon vos besoins, ajoutez de nouvelles vidéos, éventuellement d'autres attributs pour les décrire, etc.

Vous connaissez maintenant les différents types de variables, simples et structurés, qui vous permettront de décrire vos données et de les utiliser. Vous verrez dans les chapitres suivants comment en faire un usage plus intéressant et un peu moins simpliste…

Chapitre 4

Contrôler un programme

Un programme n'est pas qu'une suite directe de lectures et d'affectations de variables, même s'il y a beaucoup de cela. De plus, si l'on devait faire tout le reste à la main, cela limiterait l'intérêt d'un ordinateur et de sa puissance de calcul. C'est pourquoi vous allez voir dans ce chapitre comment structurer un programme et le contrôler, c'est-à-dire lui donner un certain comportement selon différents critères. Il sera question de conditions, de boucles, de fonctions, de procédures, etc.

4.1. Imposer des conditions

Nous allons traiter ici des instructions conditionnelles. Elles permettent d'exécuter une série d'instructions selon le résultat d'un test.

La plus utilisée d'entre elles est le Si… Alors… Sinon, dit aussi `If… Then… Else`. Elle permet d'exécuter un bloc d'instructions selon le résultat d'un test booléen, dont la réponse est vraie ou fausse. Sa syntaxe générale est :

```
If (condition) Then
  InstructionsThen
Else
  InstructionsElse
End If
```

Cette expression signifie que si `condition` est vrai, alors `InstructionsThen` sera exécuté, et dans le cas contraire, le programme exécutera `InstructionsElse`. `condition` peut être de multiples natures, toutes valables à partir du moment où le résultat est une valeur booléenne. Ce peut être :

- une variable booléenne : `If (il_pleut)` ;
- une expression comparative : `If (3 + 2 == 5)` ;
- une fonction booléenne : `If (il_fait_beau())`.

La partie `Else` de la condition est optionnelle. Si vous souhaitez exécuter des instructions dans des conditions particulières, et ne rien faire dans le cas contraire, il vous suffit de retirer la partie `Else`. Cela donne :

```
If (condition) Then
  InstructionsThen
End If
```

Ici, si `condition` est vrai, le programme exécutera `InstructionsThen`, mais dans le cas contraire, il continuera son

exécution normalement. Les parenthèses autour de la condition ne sont pas obligatoires, mais il est fortement conseillé de prendre l'habitude de les mettre. En effet, si la condition est assez longue, ou composée de plusieurs conditions, elles favorisent la compréhension.

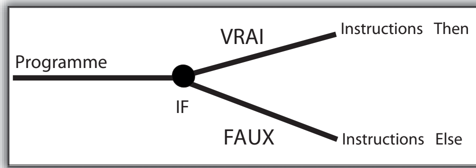


Figure 4.1 : *Représentation de la condition If… Then… Else*

La clause `ElseIf` existe dans l'écriture des instructions conditionnelles pour imposer une condition dans le cas où la première serait fausse. Exemple :

```
Dim il_fait_beau As Boolean
Dim il_pleut As Boolean
If (il_fait_beau) Then
  MessageBox.Show("Il faut prendre une casquette.")
ElseIf (il_pleut) Then
  MessageBox.Show("Il faut prendre un parapluie.")
EndIf
```

Ces lignes stipulent qu'il faut prendre une casquette s'il fait beau, et s'il pleut, que l'on doit prendre un parapluie. Le fait d'imposer la seconde condition fera qu'il n'y aura rien de spécial en cas de grisaille. Si l'on avait utilisé un `Else` sans préciser la condition sur la pluie, le programme aurait dit de prendre un parapluie dans tous les cas, sauf par beau temps.

Lorsque qu'une condition porte sur une variable avec plusieurs possibilités, il est possible d'écrire une série d'expressions conditionnelles correspondant à ces différentes possibilités. On parle alors d'"imbrication" :

```
Dim age As Integer
If (age < 16) Then
  MessageBox.Show("Vous êtes trop jeune pour travailler.")
Else
  If (age > 64) Then
    MessageBox.Show("Vous ne pouvez plus travailler.")
  Else
    MessageBox.Show("Vous avez l'âge légal pour travailler")
  EndIf
EndIf
```

On pourrait, dans ce cas, utiliser un `ElseIf`, mais cet exemple sert à montrer le principe d'imbrication. Attention, un excès d'instructions conditionnelles imbriquées peut poser des problèmes de compréhension. De plus, en termes de performances, ce n'est généralement pas la meilleure solution. À utiliser donc avec prudence !

4.2. Faire les bons choix

L'instruction Select

Après le `If… Then… Else`, qui permet d'exécuter des instructions selon une condition particulière, vous allez découvrir le `Select`, qui permet de séparer des instructions selon la valeur d'une variable donnée. Sa syntaxe est la suivante :

```
Select Case variable
  Case Valeur1
    Instructions1
  Case Valeur2
    Instructions2
  …
  Default
    InstructionsDefault
End Select
```

Cette expression signifie que, selon les valeurs de `variable`, certaines instructions seront exécutées. En particulier, si `variable` vaut `Valeur1`, le programme exécutera `Instructions1`, si `variable` vaut `Valeur2`, `Instructions2` sera exécuté, et si `variable` vaut une valeur non prise en charge dans l'expression, `InstructionsDefault` sera exécuté.

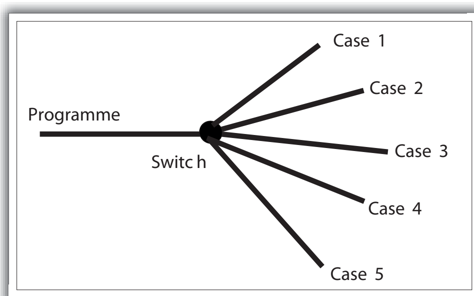


Figure 4.2 : *Représentation de l'instruction Select*

N'importe quel type simple peut être utilisé comme variable à partir du moment où le type de la variable est cohérent avec les valeurs des différents cas. Ceci provoque une erreur :

```
Dim age As Integer
Select Case age
  Case "mineur"
  Case "majeur"
End Select
```

La variable est de type entier alors que les valeurs proposées sont des chaînes de caractères. La comparaison ne pouvant être faite, cette expression est fausse et provoque une erreur.

Les énumérations sont souvent combinées avec les `Select`. En effet, du fait qu'elles ont un nombre fini de valeurs, généralement significatives, elles se prêtent bien à cette instruction. Exemple :

```
Public Enum Jour
  Lundi
  Mardi
  Mercredi
  Jeudi
  Vendredi
  Samedi
  Dimanche
End Enum

Dim jour As Jour
Select Case jour
  Case Jour.Dimanche
    MessageBox.Show("C'est le week-end et les magasins
    ✂ sont fermés.")
  Case Jour.Samedi
    MessageBox.Show("C'est le week-end mais les magasins
    ✂ sont ouverts.")
  Default
    MessageBox.Show("C'est la semaine. Vous devez aller
    ✂ travailler.")
End Select
```

Nous avons vu les principales instructions conditionnelles, qui permettent de faire telle ou telle chose dans telle ou telle situation, ce qui les rend plus intéressants.

4.3. Répéter des opérations

À présent, vous allez apprendre à "boucler", c'est-à-dire à répéter des opérations un certain nombre de fois, sans avoir à intervenir.

Les boucles représentent l'un des mécanismes les plus importants de la programmation, si ce n'est le plus important. En effet, elles permettent de tirer le meilleur parti de l'ordinateur et de sa puissance de calcul. Imaginez que vous voulez envoyer un même e-mail à un grand nombre de personnes. Ce serait une tâche horrible à exécuter à la main. En revanche, si vous écrivez le modèle et faites une boucle qui l'envoie à tous les destinataires de votre Carnet d'adresses, cela n'aura pris que le temps d'écrire le programme. C'est ce qui rend les boucles aussi importantes en programmation.

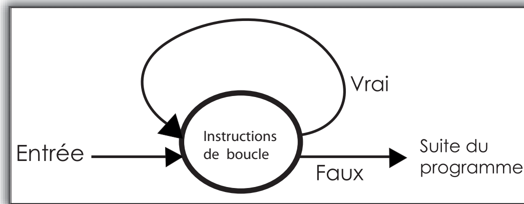


Figure 4.3 : *Fonctionnement d'une boucle*

Les boucles, ou instructions répétitives, peuvent être de deux types :

- Les boucles non déterministes : le nombre d'itérations (tours de boucle) est non défini à l'avance. Ce sont les boucles Tant que… Faire.
- Les boucles déterministes : le nombre d'itérations est connu car on spécifie un état de départ et un état d'arrêt. Ce sont les boucles Pour… Faire.

La boucle Tant que… Faire

La boucle Tant que… Faire est non déterministe, c'est-à-dire qu'il n'est pas indiqué dans la boucle elle-même le nombre d'itérations qui seront faites. Voici sa syntaxe :

```
While (condition_continuite)
  Instructions
End While
```

La boucle commence avec le mot-clé `While` et se termine au niveau du `End While`. `Instructions` correspond aux instructions qui seront exécutées lorsque l'on passera dans cette boucle. Comme dans un `If`, `condition_continuite` est un test booléen qui fera que l'on passe ou pas dans la boucle. Si sa valeur est vraie, on passera (ou repassera) dedans, et si elle est fausse, on la quittera et le programme continuera son exécution normalement. Comme dans le `If`, `condition_continuite` peut être :

- une variable booléenne : `If (il_pleut)` ;
- une expression comparative : `If (3 + 2 == 5)` ;
- une fonction booléenne : `If (il_fait_beau())`.

Voici un exemple de boucle basique, à savoir un compteur :

```
Dim compteur As Integer = 0
While (compteur < 100)
  compteur = compteur + 1
  MessageBox.Show("Compteur = " + compteur)
End While
```

Cet exemple compte de 0 à 99 et affiche la valeur courante dans une boîte de dialogue. Examinons les principaux éléments de cette boucle.

Remarquez l'importance de l'état de départ. En effet, s'il était inconnu à l'entrée dans la boucle, on ne pourrait être sûr de la bonne exécution de cette dernière. C'est pourquoi on a initialisé `compteur`. De cette manière, on est sûr que la condition de continuité au départ sera respectée et que la boucle sera donc exécutée.

L'autre élément important est la modification de la variable de boucle : `compteur`. La continuité en dépend directement. L'une des instructions de boucle modifie `compteur`. Elle est indispensable au bon fonctionnement de la boucle. C'est même ici l'instruction la plus importante. Imaginez ce qu'il se passerait si `compteur` n'est pas modifié :

1. `compteur` vaut 0. On entre dans la boucle.
2. On affiche sa valeur. On revient au début de la boucle.
3. `compteur` vaut toujours 0. On entre de nouveau dans la boucle.
4. On réaffiche sa valeur, qui est toujours la même d'ailleurs. Et l'on revient au début.
5. `compteur` vaut encore 0. On entre une nouvelle fois dans la boucle, et ainsi de suite, sans fin.

On vient de créer une boucle infinie qui affiche à chaque fois 0 dans une boîte de dialogue. C'est pourquoi il est indispensable de bien modifier la variable de boucle pour garantir une sortie. Cependant, il ne faut pas la modifier n'importe comment non plus, car cela pourrait provoquer une sortie inopinée. Si vous utilisez la variable de boucle à mauvais escient, vous pouvez provoquer des comportements non voulus du programme. Exemple :

```
Dim compteur As Integer = 0
While (compteur < 100)
  compteur = compteur + 1
  If (compteur == 14) Then
    compteur = 25
End If
  MessageBox.Show("Compteur = " + compteur)
End While
```

En ajoutant une instruction, on provoque un trou entre 14 et 25. Ici, cela est détectable. Mais des modifications peuvent être moins évidentes que celle-ci à repérer. La règle d'or est de ne jamais utiliser une variable de boucle pour autre chose que compter le nombre d'itérations et de toujours modifier sa valeur à chaque tour pour éviter une boucle infinie.

La boucle Faire… Tant que

Faire… Tant que, ou `Do… Loop While`, est comparable à Tant que… Faire, à ceci près que la condition de continuité est appliquée à la fin de la boucle, et non au début. Voici la syntaxe :

```
Do
  Instructions
Loop While (condition_continuite)
```

Si, une fois que `Instructions` est exécuté, `condition_continuite` est vrai, il y a un nouveau tour de boucle ; sinon, la boucle est terminée et le programme continue son exécution normalement. On retrouve exactement les mêmes éléments que dans la boucle Tant que… Faire. Nous n'y reviendrons pas. Pour bien comprendre la différence, considérez ces deux exemples :

```
Dim compteur As Integer = 0
While (compteur == 1)
  compteur = compteur + 1
MessageBox.Show("Compteur = " + compteur)
End While

Dim compteur As Integer = 0
```

```
Do
  Compteur = compteur + 1
  MessageBox.Show("Compteur = " + compteur)
Loop While (compteur == 1)
```

Dans les deux programmes, toutes les composantes sont les mêmes : une boucle, avec la même condition de continuité, les mêmes instructions de boucle et les mêmes états initiaux.

Dans le premier cas, `compteur` vaut 0 lorsqu'on entre dans la boucle. La première condition n'étant pas vérifiée, le programme ne passe pas dans la boucle.

Dans le deuxième cas, `compteur` vaut également 0. Le programme passe dans la boucle, change la valeur de `compteur`, et affiche "Compteur = 1". La condition de continuité est alors vérifiée. Le programme repasse donc dans la boucle. Le programme augmente `compteur` et affiche "Compteur = 2". La condition de continuité n'est alors plus vérifiée, et le programme sort de la boucle.

Dans le second cas, on effectue deux passages, alors que dans le premier, on n'entre même pas dans la boucle. C'est ce qui fait la différence entre les deux, alors que les conditions de continuité et les instructions de boucle sont les mêmes. À peu de choses près, ces boucles sont identiques, mais il y a néanmoins quelques subtilités à connaître pour éviter les petits désagréments.

La boucle Pour… Faire

Pour… Faire est une boucle déterministe. Elle ressemble à Faire… Tant que et à Tant que… Faire dans le sens où elle possède également une condition de continuité et des instructions de boucle qui seront exécutées lors du passage dans ladite boucle. La différence ici est que l'on impose un intervalle de boucle, c'est-à-dire que l'on donne à l'avance la valeur de départ de la variable de boucle, ainsi que sa valeur de fin. Il faut donc connaître au préalable le nombre de fois que l'on va boucler. Voici la syntaxe :

```
For variable_de_boucle = valeur_de_depart to valeur_de_fin
  Instructions
Next
```

D'après cette construction, on voit que l'on ne peut utiliser que des variables de boucle de type numérique. En effet, contrairement aux cas

précédents, il n'y a pas de condition de continuité booléenne qui permettrait d'utiliser n'importe quel type d'expression pour gérer les passages dans la boucle. Celle-ci s'exécutera jusqu'à ce que la variable de boucle atteigne sa valeur finale. Il n'est pas nécessaire de la modifier. C'est l'instruction `Next` qui se charge de l'augmenter. De cette manière, on ne risque pas de rencontrer le problème de boucle infinie. C'est pourquoi il est recommandé d'utiliser tant que possible la boucle `For`, dont le comportement est moins hasardeux, justement à cause de sa nature déterministe. Voici l'équivalent de l'exemple précédent si l'on utilise un `For` :

```
Dim compteur As Integer
For compteur = 0 to 100
  MessageBox.Show("Compteur = " + compteur)
Next
```

Attention, toutes les valeurs entre les bornes (bornes comprises) seront utilisées. En d'autres termes, le programme fait ici 101 tours de boucle, de 0 à 100 compris. Dans les autres boucles, étant donné que la condition d'arrêt est `(compteur < 100)`, le programme n'affiche pas "Compteur = 100", alors que, cette fois, il l'affiche. Par défaut, la variable de boucle augmente par pas de 1. Cependant, il est possible de l'augmenter d'un pas plus grand grâce au mot-clé `Step`, qui permet de spécifier une valeur de pas :

```
Dim compteur As Integer
For compteur = 0 to 100 Step 2
  MessageBox.Show("Compteur = " + compteur)
Next
```

Ici, la variable de boucle augmente par pas de 2, et le programme affiche les nombres pairs entre 0 et 100. On peut aussi faire des boucles décroissantes :

```
Dim compteur As Integer
For compteur = 100 to 0 Step -1
  MessageBox.Show("Compteur = " + compteur)
Next
```

Dans cet exemple, l'affichage des valeurs se fait dans l'ordre décroissant, de 100 à 0.

La boucle `For` est certes moins permissive et moins flexible à cause de l'absence de condition de continuité booléenne, mais elle permet d'éviter des erreurs d'utilisation à cause d'une condition mal écrite. Il est recommandé de privilégier son utilisation par rapport aux autres boucles.

4.4. Bien découper un programme

Les programmes examinés jusqu'à présent se limitent à des suites d'instructions qui s'exécutent au fur et à mesure. Cependant, on a fréquemment besoin de réutiliser des lignes de code qui ont déjà été exécutées auparavant, c'est-à-dire de refaire un même traitement. Il serait dommage de récrire ces lignes encore et encore. En ce sens, nous allons voir ici comment structurer un programme et créer du code réutilisable à plusieurs endroits. Pour y parvenir, nous utiliserons les fonctions et les procédures, et tout ce qu'il y a autour, à savoir les paramètres, les valeurs de retour, etc.

Les fonctions et procédures

Les fonctions et les procédures sont les éléments primordiaux des programmes, lorsqu'ils commencent à devenir un peu conséquents. Elles permettent de regrouper une ou plusieurs instructions. C'est en quelque sorte un sous-programme, à ceci près qu'il n'est pas autonome et qu'il est utilisé par le programme principal à un certain moment. On dit que le programme appelle une fonction ou une procédure.

Une procédure est série d'instructions regroupées, généralement utilisées pour l'affichage ou la sauvegarde, car elles n'ont pas de valeur de retour. On utilise le mot-clé `Sub` pour la déclarer, suivi du nom de la procédure, de paramètres entre parenthèses (que nous verrons plus tard) et de `End Sub` en fin de déclaration :

```
Public Sub HelloWorld()
  MessageBox.Show("Bonjour")
  MessageBox.Show("le")
  MessageBox.Show("monde")
End Sub
```

Le nom de la procédure sert à l'appeler à partir d'un autre endroit du programme. `HelloWorld` regroupe trois instructions, qui affichent chacune un mot de la phrase "Bonjour le monde" dans une boîte de dialogue. On l'appelle dans le programme tout simplement en indiquant son nom et des paramètres, s'il y en a :

```
HelloWorld()
```

À l'endroit de votre programme où vous inscrivez ce nom, c'est comme si vous récriviez les instructions regroupées dans la procédure. Vous le pouvez l'appeler autant de fois que nécessaire dans le programme.

Écrire des procédures permet de gagner en compréhension et en temps, car vous n'avez pas besoin de récrire toutes les instructions à chaque fois. Ici, en un appel, vous avez récrit l'équivalent de trois instructions, mais les procédures peuvent être beaucoup plus longues.

Une fonction est une série d'instructions regroupées, comme dans une procédure, mais elle permet de renvoyer une valeur de retour. Cette valeur peut être utilisée par le programme appelant comme si c'était une valeur normale. Voici comment déclarer une fonction :

```
Public Function HelloWorld()
  Dim resultat As String = "Bonjour le monde"
Return (resultat)
End Function
```

`HelloWorld` renvoie tout simplement la chaîne de caractères "Bonjour le monde". Vous n'êtes pas obligé de préciser le type de la valeur de retour mais vous pouvez le faire en ajoutant `As` et le type après le nom et les paramètres. Il est recommandé de le préciser, car le code est plus clair. Cela aide à la compréhension et permet de détecter plus facilement les erreurs.

```
Public Function HelloWorld() As String
  Dim resultat As String = "Bonjour le monde"
Return (resultat)
End Function
```

`Return` est le mot-clé qui sert à spécifier que l'on sort de la fonction en retournant la valeur de retour. On peut également affecter la valeur de retour au nom de la fonction, comme ceci :

```
Public Function HelloWorld() As String
  Dim resultat As String = "Bonjour le monde"
HelloWorld = resultat
End Function
```

Les deux fonctions sont exactement les mêmes et, bien que ce ne soit pas d'une importance primordiale, mieux vaut le savoir pour comprendre des programmes écrits par d'autres.

Vous pouvez appeler une fonction comme une procédure. Dans ce cas, la valeur de retour n'a pas d'effet, mais les instructions de la fonction sont quand même exécutées :

```
HelloWorld()
```

La fonction se contente de renvoyer une chaîne de caractères. Mais vous pouvez utiliser cette valeur de retour dans n'importe quelle partie du programme qui attend une valeur du même type. Exemple :

```
MessageBox.Show(HelloWorld())
```

Par cet appel, le programme affiche une boîte de dialogue contenant le message "Bonjour le monde", car `Show` attend une chaîne de caractères, et justement, `HelloWorld` en renvoie une.

Les paramètres

Lorsque l'on déclare ou que l'on appelle des fonctions ou des procédures, leur nom est toujours suivi de parenthèses vides ou qui contiennent des paramètres. Les paramètres sont des variables, dont on récupère la valeur lors de l'appel de la fonction ou de la procédure. Il peut y en avoir un, plusieurs, ou aucun. Ils permettent de donner de la flexibilité à une fonction ou à une procédure. En effet, étant donné que leur valeur exacte n'est donnée que lors de l'appel, celle-ci peut varier et apporter du changement. À la déclaration, il faut spécifier leur type et leur nom. Voici un exemple de procédure avec des paramètres :

```
Public Sub QuiSuisJe(ByVal nom As String, ByVal prenom As
✂ String)
  MessageBox.Show("Je suis " + nom + " " + prenom)
End Sub
```

Le principe est le même pour les fonctions :

```
Public Function Moi(ByVal nom As String, ByVal prenom As
✂ String) As String
  Return ("Je suis " + nom + " " + prenom)
End Function
```

Par contre, lorsque l'on appelle des fonctions ou des procédures qui ont des paramètres, il faut mettre entre parenthèses les valeurs de ces paramètres. On peut le faire soit par le biais de variable, ou d'expression constante, voire de fonction. En effet, une fonction peut être un paramètre d'une autre fonction, car elle retourne une valeur ; mais il faut que les types correspondent.

```
Dim mon_nom As String = "Martin"
Dim mon_prenom As String = "Pierre"
QuiSuisJe(mon_nom, mon_prenom)
MessageBox.Show(Moi("Martin", "Pierre"))
```

Le mot-clé ByVal qui figure dans la déclaration des paramètres indique qu'ils sont transmis par valeur. Cela signifie qu'à chaque fois qu'un paramètre est fourni, une copie des données est faite avant qu'elles soient transmises à la fonction ou à la procédure. Cela garantit qu'un changement sur la valeur transmise ne sera pas répercuté sur la variable originale. Il s'agit du comportement par défaut, ce qui rend ByVal optionnel. Modifions un peu la procédure QuiSuisJe :

```
Public Sub QuiSuisJe(ByVal nom As String, ByVal prenom As
✂ String)
  MessageBox.Show("Je suis " + nom + " " + prenom)
  nom = "DOE"
  prenom = "John"
End Sub
```

Après avoir affiché le nom et le prénom, on les modifie. On reprend le programme précédent, avec une légère modification :

```
Dim mon_nom As String = "Martin"
Dim mon_prenom As String = "Pierre"
QuiSuisJe(mon_nom, mon_prenom)
MessageBox.Show(Moi(mon_nom, mon_prenom))
```

Au lieu d'écrire des valeurs constantes dans la dernière instruction, on indique mon_nom et mon_prenom. Le programme affiche deux fois une boîte de dialogue dans laquelle figure le texte "Je suis Martin Pierre", bien que les paramètres aient été modifiés dans la deuxième procédure QuiSuisJe. Cela est dû au fait que ces paramètres ont été passés par valeur. La procédure a donc travaillé sur une copie des variables mon_nom et mon_prenom et leurs valeurs originales n'ont donc pas été modifiées.

Le passage de paramètres par valeur s'oppose au passage de paramètres par référence. Dans ce cas, la procédure ou la fonction ne travaille plus avec une copie de la variable passée en paramètre, mais directement avec celle-ci, plus précisément avec son adresse mémoire. En d'autres termes, si une modification est faite pendant la procédure ou la fonction, elle est conservée hors de celle-ci. À manipuler avec une prudence donc ! Pour spécifier un passage par référence, on utilise ByRef au lieu de ByVal. Appliquons cela à la fonction QuiSuisJe :

```
Public Sub QuiSuisJe(ByRef nom As String, ByRef prenom As
✂ String)
  MessageBox.Show("Je suis " + nom + " " + prenom)
  nom = "DOE"
  prenom = "John"
End Sub
```

Si maintenant nous exécutons le même bout de programme que précédemment, on obtient une première boîte de dialogue dans laquelle figure le texte "Je suis Martin Pierre", puis une deuxième contenant "Je suis DOE John". Le fait d'avoir passé les paramètres par référence et de les avoir modifiés dans la procédure provoque leur modification réelle dans le reste du programme. Encore une fois, manipulez le passage de paramètres par référence avec précaution.

Les variables de fonction et la portée

Dans les fonctions et les procédures vues précédemment, les comportements sont relativement simplistes, mais les opérations peuvent être beaucoup complexes. Vous pouvez par exemple utiliser des variables dont la "durée de vie" est limitée aux procédures ou fonctions dans lesquelles elles apparaissent. On dit de ces variables qu'elles sont "locales". Vous pourrez les utiliser seulement dans la fonction ou la procédure en question. Elles se déclarent et s'utilisent comme des variables normales :

```
Public Function Addition(ByVal nombre1 As Integer, ByVal
✂ nombre2 As Integer) As Integer
  Dim resultat As Integer
  Return (resultat)
End Function
```

Ici, `resultat` est une variable locale de la fonction `addition`. Elle ne peut être utilisée ailleurs dans le programme.

```
Dim somme As Integer
somme = Addition(3, 5)
MessageBox.Show("Somme = " + somme)
MessageBox.Show("Resultat = " + resultat)
```

La troisième ligne affiche une boîte de dialogue contenant "Somme = 8". On peut penser que `resultat` existe car cette variable est créée dans la fonction `addition`. Or, `resultat` est une variable locale de la fonction `Addition`. Elle n'existe donc pas pour le reste du programme. De ce fait, l'utiliser hors de `Addition` provoque une erreur et il est impossible d'exécuter la seconde instruction `Show`.

Cela nous permet d'introduire la notion de portée. La portée correspond à l'accessibilité d'une variable. La portée d'une variable locale d'une fonction ou d'une procédure est simplement cette fonction ou cette procédure. Au-delà, la variable n'est pas utilisable. Il en est de même

pour les boucles. Une variable déclarée dans une boucle n'est utilisable que dans cette boucle :

```
Dim compteur As Integer
For compteur = 1 to 100
  Dim sauvegarde As Integer
  Sauvegarde = compteur
Next
MessageBox.Show("Sauvegarde = " + sauvegarde)
```

La variable `sauvegarde` étant déclarée et utilisée dans la boucle `For`, sa portée se limite à cette boucle. C'est pourquoi l'instruction `Show` de ce bout de programme provoque une erreur. `sauvegarde` étant utilisée hors de sa portée, elle n'est pas accessible. C'est comme si elle n'existait pas, on ne peut pas s'en servir. En raison de la portée des variables, faites bien attention à l'endroit où vous les déclarez pour être sûr qu'elles existent au moment où vous vous en servez. Bien que ce principe puisse paraître subtil au premier abord, vous arriverez vite le maîtriser.

Chapitre 5

Dialoguer avec un ordinateur

Les boutons ou les *TextBox*, que vous avez utilisés dans les programmes précédents, sont des contrôles qui permettent une interaction avec l'utilisateur. Dans ce chapitre, vous allez apprendre à maximiser cette interaction, à rendre vos programmes plus vivants, à ajouter du contenu multimédia et à naviguer sur le Web. Les possibilités sont sans fin. Le principe est simple. Lorsque vous lancez Visual Basic, vous arrivez dans le designer de fenêtres. Ces fenêtres sont appelées "formulaires", car elles sont utilisées pour capturer des informations de l'utilisateur et lui donner des réponses en conséquence. Dans les chapitres précédents, vous avez par exemple double-cliqué sur un bouton pour vous placer dans l'édition de code de l'événement correspondant au clic du bouton. La programmation d'applications pour Windows est fondée sur ce simple concept : un événement, une réponse. L'utilisateur du programme clique sur un bouton ? Les instructions que vous avez prévues pour cet événement sont exécutées.

Comme vous allez le voir, chaque contrôle possède ses propres événements.

5.1. Les différents contrôles

Définition

Vous avez déjà utilisé la boîte à outils de Visual Basic Express au cours des différents chapitres, pour ajouter un bouton ou une *TextBox* à votre application. Ces éléments sont des contrôles Windows. Ils possèdent chacun des propriétés, qui sont visibles dans la fenêtre des propriétés, et des événements, qui sont visibles d'un clic sur l'icône en forme d'éclair de la fenêtre des propriétés.

Les propriétés permettent de modifier l'état de chaque contrôle. Par exemple, ouvrez un nouveau projet avec Visual Basic Express puis ajoutez un bouton en le faisant glisser sur le formulaire qui se trouve dans le designer depuis la boîte à outils. Cliquez sur le bouton et appuyez sur la touche F4. La fenêtre des propriétés s'affiche et vous pouvez modifier l'état du bouton, de son nom jusqu'à sa couleur de fond. Commençons par exemple par changer le texte de ce bouton.

1 Sélectionnez le bouton d'un simple clic.
2 Appuyez sur la touche F4.

3 Dans la liste qui s'affiche, descendez jusqu'à l'élément *Text*.

4 À droite de *Text*, entrez le texte que vous voulez, par exemple `MonBouton`.

5 Validez avec ↵. Le changement est immédiat.



Figure 5.1 : *Changement de texte du bouton*

Vous savez maintenant manipuler les propriétés d'un contrôle. Le principe est analogue quel que soit le type du contrôle puisque tous disposent de la même fenêtre de propriétés. Attention toutefois, certaines propriétés sont disponibles pour plusieurs contrôles différents, mais ne modifient pas la même chose. Par exemple, la propriété *Text* que vous venez de manipuler change le texte d'un bouton, mais si vous modifiez cette même propriété pour un formulaire, c'est son titre qui est changé.

Les événements

Chaque contrôle possède son lot d'événements modifiables de manière à donner au programme le comportement souhaité. Pour modifier un événement d'un contrôle, vous avez deux solutions. Vous avez déjà vu la première, qui consiste à double-cliquer sur le contrôle, ce qui place l'éditeur de texte dans un événement par défaut. C'est le cas du clic sur un bouton ou du changement de texte d'une *TextBox*. En revanche, pour modifier le comportement d'un programme lorsque le pointeur de la souris quitte la zone d'un bouton, il suffit d'aller dans la liste des événements en cliquant sur l'icône en forme d'éclair de la fenêtre des propriétés et de parcourir la liste. Affichons par exemple un message à chaque fois que l'utilisateur entre dans la zone d'un bouton.

1 Lancez Visual Basic Express et créez un nouveau projet.

2 Dans le designer de formulaires, ajoutez un bouton par glisser-déposer.

3 Sélectionnez le bouton en cliquant dessus puis appuyez sur F4.

4 Dans la fenêtre des propriétés, modifiez la propriété *Text* par `Ne cliquez pas ici !`.

5 Cliquez sur l'icône *Événements* de la fenêtre des propriétés (le petit éclair), puis cherchez l'événement *MouseEnter*.

6 Double-cliquez sur la case vide dans la colonne de droite qui correspond à l'événement.

7 Une fois dans l'éditeur de code, ajoutez l'instruction `MessageBox.Show("NE PAS CLIQUER !")`.

8 Lancez le programme avec F5, puis passez la souris au-dessus du bouton, sans cliquer. Le résultat est immédiat.

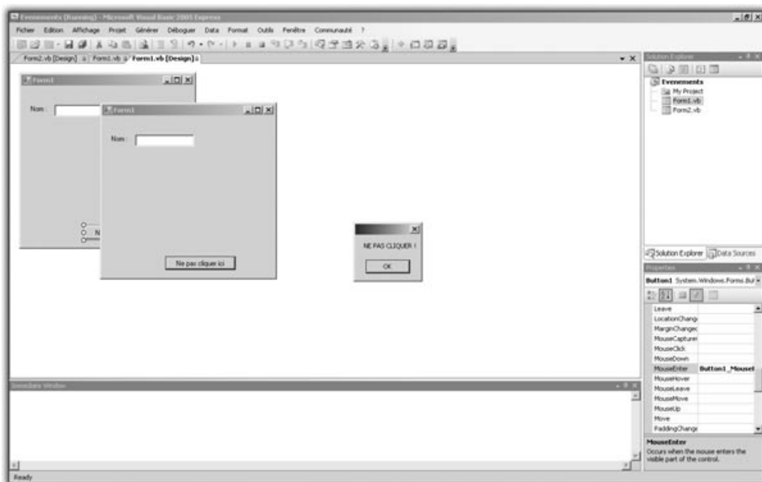


Figure 5.2 : *Résultat de l'événement MouseEnter*

Le contrôle Label

Dans la boîte à outils, faites glisser sur le formulaire le contrôle nommé *Label* ("étiquette" en anglais). Il est couramment utilisé pour spécifier à quoi correspond une zone de texte.



Figure 5.3 :
Label avec une TextBox

Dans ce cas, nous utilisons un *Label* pour guider l'utilisateur : en regardant la fenêtre, il sait tout de suite qu'il faut entrer un nom dans la *TextBox*.

Pour modifier le texte d'un *Label*, il suffit de le sélectionner en cliquant dessus puis, après avoir appuyé sur F4 pour afficher les propriétés, de modifier la propriété *Text*.

Les propriétés principales d'un *Label* sont *Text*, qui représente le contenu du *Label*, et *(Name)*, qui représente le nom du contrôle pour le programme. Vous pouvez accéder aux propriétés d'un contrôle via la fenêtre des propriétés, mais également dans le code, ce qui permet de modifier un contrôle pendant que le programme s'exécute. Voici un exemple :

1. Ouvrez un projet Visual Basic Express et ajoutez un bouton et un *Label* à votre formulaire dans le designer.
2. Dans les propriétés du *Label*, modifiez le texte par `Vous n'avez pas cliqué sur le bouton`.
3. Modifiez la propriété *(Name)* de ce *Label* par `Label_Click`.
4. Double-cliquez sur le bouton pour éditer l'événement "clic".
5. Une fois dans l'éditeur de code, ajoutez l'instruction `Label_Click.Text = "Vous avez cliqué sur le bouton"`.
6. Lancez le programme avec F5 et cliquez sur le bouton du formulaire.



Figure 5.4 :
Avant le clic



Figure 5.5 :
Après un clic sur le bouton

Le contrôle Button

Ce contrôle est l'un des plus utilisés. Dès lors que l'on veut un retour de l'utilisateur, cela passe souvent par un clic sur un bouton.

L'événement le plus courant et le plus modifié pour ce contrôle est évidemment le clic, mais vous pouvez changer le comportement associé au passage du curseur ou, à l'inverse, lorsque l'utilisateur quitte la zone du bouton sans cliquer dessus.

Les propriétés les plus utiles pour un bouton sont *Text*, qui représente le texte inscrit sur le bouton, et *(Name)*, qui représente le nom du bouton pour le programme. Vous pouvez accéder à ces propriétés (comme à celles d'un *Label*) directement depuis le code. Dans l'exemple suivant, nous allons réaliser un compteur de clics à l'aide d'un *Button* et d'un *Label*.

1 Ouvrez un nouveau projet Visual Basic Express, puis ajoutez un bouton et un *Label* au formulaire.

2 Double-cliquez sur le bouton pour éditer le code de l'événement "clic".

3 Ajoutez les instructions suivantes :

```
Dim i As Integer
i = Int32.Parse(Label1.Text)
i = i + 1
Label1.Text = i.ToString
```

4 Lancez le programme avec F5 puis cliquez sur le bouton. Le texte du *Label* est modifié en fonction du nombre de clics effectués.

Le contrôle ListBox

Ce contrôle permet de proposer une liste à l'utilisateur. Pour l'utiliser, faites-le glisser sur le formulaire. Avec une *ListBox*, vous allez réaliser un répertoire téléphonique.

Une fois le contrôle *ListBox* ajouté au formulaire, sélectionnez-le en cliquant dessus, puis appuyez sur F4 pour visualiser les propriétés. En face de la propriété *Items*, l'inscription *Collection* apparaît. Une collection est un ensemble de valeurs. Vous allez éditer cette collection avec l'outil approprié. Pour cela, cliquez sur l'icône décorée de trois points de suspension, à droite de la mention *Collection*.



Figure 5.6 : *L'éditeur de collection*

Dans la fenêtre qui apparaît, entrez un nom et un numéro de téléphone, séparés par un espace, en effectuant un retour à la ligne entre chaque saisie.

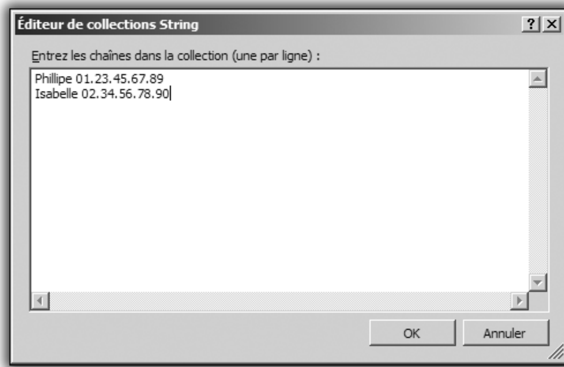


Figure 5.7 : *L'éditeur une fois rempli*

Cliquez ensuite sur le bouton OK de l'éditeur, puis lancez le programme avec F5. La fenêtre s'affiche et la *ListBox* présente les noms et numéros saisis.

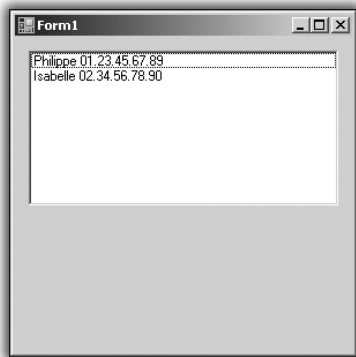


Figure 5.8 :
Le programme une fois lancé

Maintenant, il serait interessant d'ajouter des éléments à la *ListBox*. Pour cela, introduisez un bouton et une *TextBox* dans le formulaire principal.

Double-cliquez sur le bouton, puis ajoutez le code suivant dans l'éditeur :

```
ListBox1.Items.Add(TextBox1.Text)
```

Cette instruction ajoute à la collection d'objets de la *ListBox* le texte qui est saisi dans la *TextBox*.

Imaginons maintenant que vous vouliez effacer l'une des entrées de la *ListBox*. Cette opération est des plus simples.

Fermez l'application puis revenez au designer. Ajoutez à votre formulaire un deuxième bouton, qui servira à effacer une entrée. Dans les propriétés du bouton, modifiez le texte pour qu'il affiche "Suppression".

Double-cliquez ensuite sur ce deuxième bouton pour entrer dans l'éditeur de code à l'endroit du code exécuté lors d'un clic sur ce bouton.

Entrez alors l'instruction suivante :

```
ListBox1.Items.Remove(ListBox1.SelectedItem)
```

Cette ligne spécifie au programme qu'il faut retirer de la collection d'objets de la *ListBox* l'objet qui est sélectionné au moment du clic (`ListBox1.SelectedItem`). Lancez ensuite le programme avec F5 et testez la suppression via le nouveau bouton.

Vous savez maintenant comment ajouter et supprimer des objets dans une *ListBox*.



Renommer les contrôles

Vous pouvez simplifier l'édition d'un programme en renommant les contrôles à l'aide de leur propriété *name*. En renommant par exemple *Label1* en *MaLabel*, vous pourrez modifier sa propriété `Text` dans le code en saisissant `MaLabel.text`.

Les contrôles PictureBox et OpenFileDialog

Ce contrôle est simplement une boîte à image. Vous pouvez par ce biais afficher les images que vous voulez. Il dispose d'une propriété qui permet de sélectionner l'image par défaut. Elle est logiquement nommée *Image*. Vous pouvez la modifier dans la fenêtre des propriétés, qui permet de sélectionner une image sur votre ordinateur et de l'incorporer au fichier ressource. Cliquez sur les trois points de suspension à côté de la propriété pour ouvrir l'éditeur.

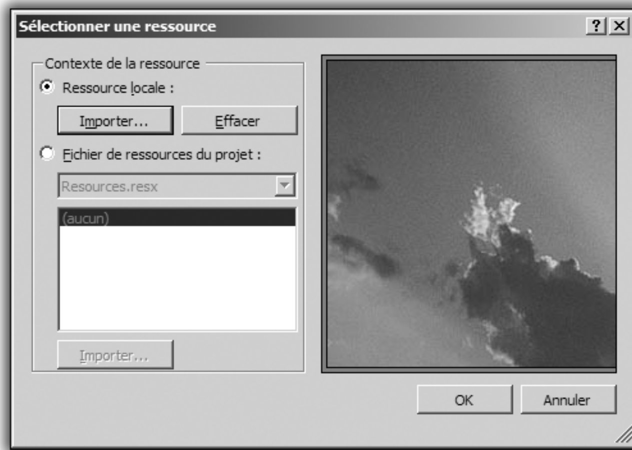


Figure 5.9 : *L'éditeur de la propriété Image*

Dans cet éditeur, sélectionnez *Ressource locale* puis cliquez sur le bouton **Importer**. Parcourez ensuite votre disque dur a la recherche d'une image qui vous plaît, puis double-cliquez dessus. L'image s'affiche dans la *PictureBox*. Mais si cette image est plus grande que la boîte insérée dans le formulaire, elle ne sera pas entièrement affichée. Pour remédier à cela, dans la fenêtre des propriétés, descendez dans la liste jusqu'à trouver la propriété *SizeMode*, puis changez-la pour la valeur *StretchImage*, qui permet d'ajuster automatiquement la taille de l'image à la taille de la *PictureBox*.

Lancez maintenant le programme avec la touche F5. La fenêtre principale se lance et contient l'image que vous avez sélectionnée. Cependant, si vous devez éditer le code du programme à chaque fois que vous voulez changer l'image affichée, c'est contraignant. Ne serait-il pas pratique d'avoir un bouton un peu spécial, qui permette de sélectionner un fichier et de le traiter en conséquence ?

Le contrôle *OpenFileDialog* est justement fait pour cela. Il est un peu particulier en ce sens qu'il permet à un utilisateur de choisir un fichier en parcourant le disque dur et de traiter le fichier en question.

Faites glisser un contrôle *OpenFileDialog* de la boîte à outils vers le formulaire. Vous pouvez constater qu'il ne s'ajoute pas au formulaire, comme les autres contrôles, mais qu'il va se nicher en dessous.

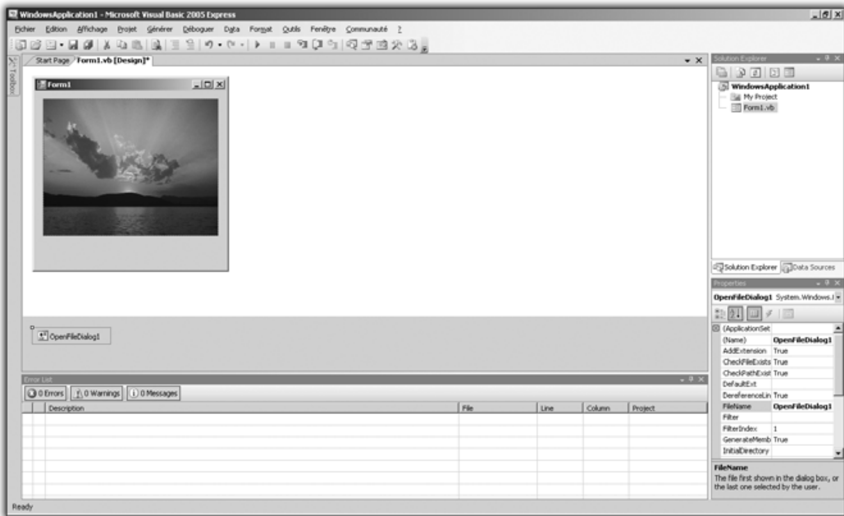


Figure 5.10 : *Le contrôle OpenFileDialog*

En fait, ce contrôle ne se manipule pas directement, mais au travers d'autres contrôles, comme un bouton par exemple. Pour illustrer le fonctionnement, ajoutez un bouton à votre application puis double-cliquez dessus pour afficher le code de l'événement "clic". Ajoutez la ligne d'instruction `OpenFileDialog1.ShowDialog()`. Ensuite, lancez votre programme en appuyant sur la touche F5. Une fois le programme lancé, cliquez sur le bouton. Une boîte de dialogue qui vous propose de choisir un fichier s'affiche.

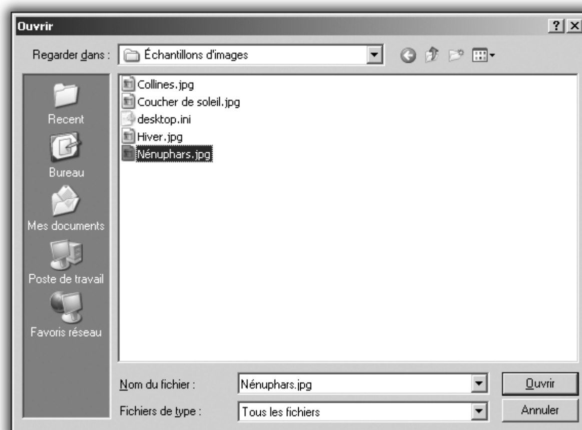


Figure 5.11 : *La boîte de dialogue*

Toutefois, lorsque vous sélectionnez un fichier à l'aide de cette boîte de dialogue, il ne se passe rien. Pour remédier à cela, ajoutez le code suivant à l'événement "clic" :

```
PictureBox1.Image = Image.FromFile(OpenFileDialog1
✂ .FileName)
```

Cette ligne demande simplement à la *PictureBox* de changer l'image en en construisant une à partir d'un fichier sélectionné via la boîte de dialogue générée par le contrôle *OpenFileDialog*.

Lancez le programme pour vérifier que tout fonctionne et admirez le fruit de votre programmation.

Vous disposez maintenant d'une visionneuse d'image, par exemple pour regarder les clichés issus d'un appareil photo numérique branché à votre ordinateur. Mais ne serait-il pas intéressant de choisir les fichiers dans le répertoire *Mes Images* au lieu de chercher dans le répertoire à partir duquel vous lancez l'application ? Ce serait un plus indéniable.

Pour cela, il existe la propriété *InitialDirectory* du contrôle *OpenFileDialog*. Elle ne peut pas être directement modifiée dans la fenêtre des propriétés. Vous devez intervenir sur le code. Pour cela, double-cliquez sur le formulaire pour éditer le code lancé lors du chargement de la fenêtre. Une fois dans l'éditeur de code, saisissez cette instruction :

```
OpenFileDialog1.InitialDirectory = Environment
✂ .SpecialFolder.MyPictures
```

Elle permet d'éditer le répertoire par défaut dans lequel chercher les images et de spécifier que ce répertoire sera *Mes Images*, créé par Windows pour chaque utilisateur de l'ordinateur. Il existe également le même genre de raccourci pour le répertoire *Mes Documents*, *Ma Musique*, le Bureau, etc.



Figure 5.12 :
Le programme avec l'image une fois changée

Vous êtes maintenant capable de sélectionner un fichier à l'aide d'une boîte de dialogue.

Le contrôle WebBrowser

Le contrôle *WebBrowser* permet de développer en un temps record des applications s'appuyant sur la puissance du Web. Grâce à lui, vous pouvez disposer d'un navigateur Internet à l'intérieur de votre application.

Vous allez vous en servir pour développer rapidement un navigateur Internet personnalisé.

Ouvrez un nouveau projet avec Visual Basic Express. Une fois dans le designer, ajoutez une *TextBox* dont vous changerez la propriété *(name)* en `address`, puis un contrôle *WebBrowser* que vous placerez sur le formulaire.

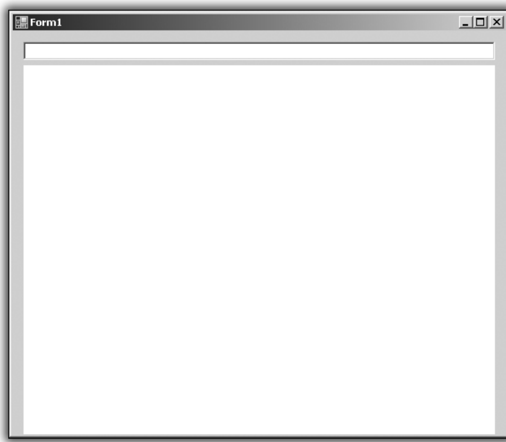


Figure 5.13 : *L'ébauche du navigateur*



Redimensionner un formulaire

N'hésitez pas à redimensionner le formulaire à l'aide de la souris en plaçant le curseur sur l'un des coins et en bougeant la souris tout en maintenant le bouton gauche enfoncé.

Ajoutez un bouton à droite de la *TextBox* puis changez la propriété *Text* du bouton par `Go`. Modifiez le code de l'événement "clic" en double-cliquant sur ce dernier et ajoutez l'instruction suivante :

```
WebBrowser1.Navigate(address.Text)
```

Lancez votre programme en appuyant sur F5. Saisissez une adresse dans la *TextBox* puis cliquez sur le bouton **GO**. Vous avez maintenant votre propre navigateur.



Figure 5.14 : *Votre navigateur terminé*

Vous avez, en 5 minutes, réalisé une application fonctionnelle, d'une qualité quasiment égale à celle d'une application professionnelle. Est-il encore possible de dire que la programmation est réservée à une élite d'informaticiens ?

Les contrôles FontDialog et ColorDialog

Vous allez apprendre à réaliser un éditeur de texte simple permettant de saisir du texte, de changer sa police et sa couleur.

Commencez par créer un nouveau projet avec Visual Basic Express. Puis, dans l'éditeur de formulaire, ajoutez deux boutons, un contrôle *TextBox* et les contrôles *FontDialog* et *ColorDialog*. Comme le contrôle *OpenFileDialog* que vous avez manipulé précédemment, ces deux derniers contrôles ne viennent pas s'ajouter au formulaire, mais juste en dessous. Il faudra donc utiliser les deux boutons pour faire le lien entre les contrôles et leur utilisation. Renommez le texte d'un des boutons en `Police` et le texte du second bouton en `Couleur`. Double-cliquez sur le bouton **Police** et ajoutez les instructions suivantes au code de l'événement "clic" :

```
FontDialog1.ShowDialog()
TextBox1.Font = FontDialog1.Font
```

Revenez à l'éditeur de formulaire. Double-cliquez sur le bouton **Couleur** pour éditer l'événement "clic". Ajoutez les instructions suivantes :

```
ColorDialog1.ShowDialog()
TextBox1.ForeColor = ColorDialog1.Color
```

Lancez le programme avec la touche F5, saisissez du texte dans la *TextBox*, puis utilisez les deux boutons pour modifier la couleur et la police du texte.

Les instructions se limitent ici à appliquer une police au texte sélectionnée via la boîte de dialogue **Font**. Quand vous cliquez sur OK, la police choisie remplace celle de la *TextBox*. Quand vous cliquez sur l'autre bouton, il se passe exactement la même chose, mais la couleur sélectionnée dans la boîte de dialogue **Couleurs** s'applique au premier plan de la *TextBox*, à savoir au texte.



Soyez professionnel !

Pour donner un look plus professionnel à cet éditeur de texte, sélectionnez la *TextBox* d'un simple clic, puis cliquez sur son petit triangle en haut à droite. Cochez la case *Multiline*. Il est maintenant possible de modifier la hauteur en nombre de lignes de cette *TextBox*.

Vous pouvez appliquer un changement de couleur à tous les éléments de votre formulaire. Par exemple, vous pouvez accorder la couleur de fond de l'application avec la couleur de texte que vous venez de sélectionner en ajoutant la ligne suivante à l'événement approprié :

```
Me.BackColor = ColorDialog1.Color
```

`Me` fait référence à la fenêtre qui est en cours d'exécution. Vous appliquez ici la couleur à l'arrière-plan de la fenêtre. Ainsi, le texte que vous allez saisir sera de la même couleur. La méthode de changement de couleur fonctionne avec n'importe quel contrôle disposant d'une propriété *Color*. Pour vérifier que le contrôle en possède une, il suffit de regarder dans la liste de la fenêtre des propriétés. La deuxième méthode consiste à saisir le nom du contrôle dans l'éditeur de code, suivi d'un point, puis d'appuyer sur les touches Ctrl+Barre d'espace pour lister toutes les propriétés et méthodes des contrôles.

Le contrôle TreeView

Si vous vous êtes déjà servi de l'Explorateur de fichiers Windows, vous avez dû remarquer que les répertoires se présentent sous la forme d'une arborescence. Il est possible d'ajouter ce style d'arborescence dans vos programmes à l'aide du contrôle *TreeView*. Il permet d'organiser des données sous forme de nœuds hiérarchiques. Le nœud de premier niveau s'appelle "nœud racine" et les nœuds de niveaux inférieurs sont les fils du nœud racine, chaque nœud fils pouvant lui-même servir de nœud racine. Un exemple sera certainement plus parlant.

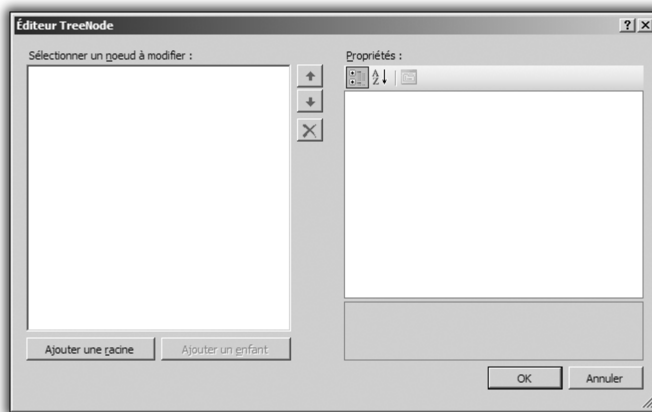


Figure 5.15 : *L'éditeur de nœud*

Créez un nouveau projet, puis ajoutez au formulaire un contrôle *TreeView*. Une fois le *TreeView* ajouté, sélectionnez-le en cliquant dessus, puis affichez la fenêtre des propriétés en appuyant sur F4. L'une des propriétés s'appelle *Nodes*. Comme la collection *Items* du contrôle *ListBox*, il s'agit d'une collection de valeurs, qui vont représenter cette

fois des nœuds de l'arborescence. Comme pour la *ListBox*, vous pouvez éditer la collection de nœuds graphiquement, en cliquant sur les trois points de suspension à droite de *Nodes*, dans la fenêtre des propriétés.

Cliquez sur le bouton **Ajouter une racine** pour ajouter un premier nœud au *TreeView*. Une fois ce nœud ajouté, vous pouvez lui adjoindre soit des enfants, soit une autre racine. Mais encore une fois, vous pouvez modifier un *TreeView* avec le code de l'application.

L'objectif ici est de créer une bibliothèque de films.

En ce sens, ajoutez à votre application trois *TextBox* et deux boutons.



Figure 5.16 : *Squelette de l'application*

Nommez le premier bouton `Ajout Catégorie` et le second `Ajout Film`. Il faut maintenant modifier les événements "clic" pour que les boutons ajoutent chacun une catégorie à un niveau de hiérarchie différent.

Double-cliquez sur le bouton **Ajout Catégorie** de manière à vous placer dans l'éditeur de code, puis recopiez l'instruction suivante :

```
TreeView1.Nodes.Add(TextBox1.Text)
```

Cette instruction ajoute un nœud au niveau hiérarchique le plus haut du *TreeView*. Double-cliquez sur le bouton **Ajout Film** et, une fois dans l'éditeur de code, saisissez l'instruction suivante :

```
TreeView1.SelectedNode.Nodes.Add(textbox2.Text)
```

Cette ligne ajoute un nœud en considérant le nœud sélectionné comme racine d'un arbre.

Lancez maintenant l'application en appuyant sur F5. Entrez une catégorie dans la première *TextBox* puis cliquez sur le bouton **Ajout Catégorie**. Sélectionnez ensuite le nœud que vous venez d'ajouter, puis saisissez un titre de film dans la deuxième *TextBox*. Cliquez ensuite sur le deuxième bouton et le nom du film viendra s'ajouter à la hiérarchie.

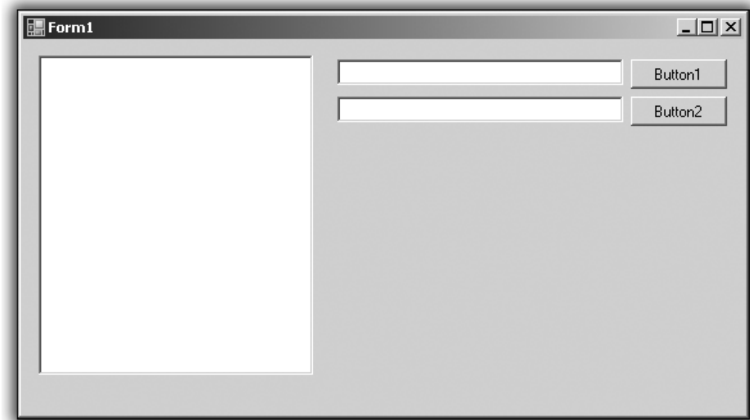


Figure 5.17 : *Votre bibliothèque de films*

Le contrôle ComboBox

Ce contrôle permet d'éviter bon nombre de saisies répétées en proposant une liste déroulante de choix possibles. Vous allez tester son utilité en remplaçant le bouton **Ajout Catégorie** de la bibliothèque de films par une *ComboBox*. Créez un nouveau projet et ajoutez au formulaire les contrôles suivants : un *TreeView*, deux *Label*, une *TextBox*, une *ComboBox* et un bouton.

Nommez le premier *Label* `Catégories` et placez-le au-dessus de la *ComboBox*. Nommez le deuxième *Label* `Film` et placez-le au-dessus de la *TextBox*. Placez le bouton juste en dessous.

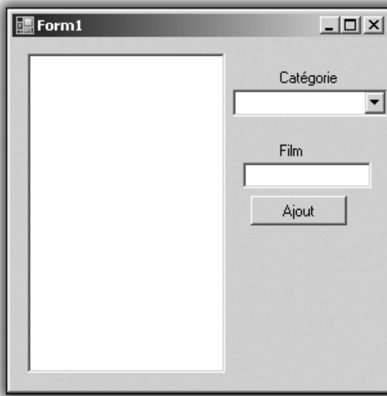


Figure 5.18 : *Bibliothèque améliorée*

Sélectionnez maintenant la *ComboBox* et affichez ses propriétés en appuyant sur F4. Dans la liste, vous retrouvez une propriété *Items* éditable, comme pour la *ListBox*. Le fonctionnement de ces deux contrôles est par bien des points similaire. Pour éditer le contenu de la *ComboBox*, cliquez sur les trois points de suspension à droite de la propriété *Items*.

Une fenêtre s'affiche alors.

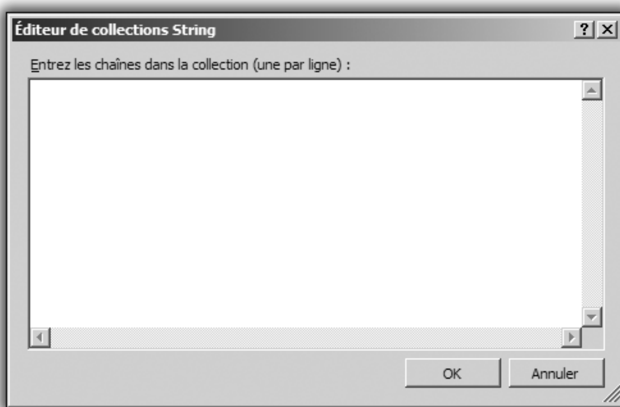


Figure 5.19 : *Ajoutez des éléments*

Ajoutez des catégories de films, une par ligne, et appuyez sur OK. Votre liste de genres est gardée dans la collection *Items* de la *ComboBox*. Vous allez maintenant modifier le code de l'événement "clic" du bouton pour ajouter un film à la liste.

En ce sens, placez-vous dans l'éditeur de code et saisissez le code suivant :

```
Dim n As TreeNode
n = New TreeNode
n.Text = ComboBox1.SelectedItem
n.Nodes.Add(New TreeNode(TextBox1.Text))
TreeView1.Nodes.Add(n)
```

Cette portion de code déclare une variable de type `TreeNode` (nœud d'un arbre) et fait en sorte que le texte affiché sur le *TreeView* soit celui de la catégorie choisie dans la *ComboBox*. Ensuite est ajouté, en tant que nœud enfant du nœud `n`, un nœud qui porte le nom du film que l'on va saisir dans la *TextBox*.



Figure 5.20 :
Le résultat de votre travail

5.2. Les formulaires

Les formulaires permettent à l'utilisateur de saisir des informations et également de lui en renvoyer en utilisant les différents contrôles mis à disposition par Visual Basic Express. Jusqu'à présent, vous avez développé des applications qui utilisaient un seul formulaire. Vous allez maintenant utiliser les formulaires pour organiser au mieux une application.

Lorsque vous créez un projet sous Visual Basic Express, un formulaire nommé *Form1.vb* est ajouté au projet. Si vous développez des applications complexes, vous aurez souvent recours à d'autres fenêtres à l'intérieur de votre programme. Pour cela, vous devrez ajouter des

formulaires au projet. Pour ajouter un formulaire, cliquez du bouton droit dans l'Explorateur de solutions situé au-dessus de la fenêtre des propriétés, puis cliquez sur **Ajouter/Nouvel élément**. Dans la liste de choix qui est alors proposée, choisissez *Formulaire Windows* (Windows Form). Un nouvel onglet, nommé *Form2.vb*, est alors ajouté au projet. Les manipulations possibles sur ce nouveau formulaire sont exactement les mêmes que sur le formulaire qui est ajouté par défaut à la création d'un nouveau projet.

Pour illustrer ce principe, reprenez la bibliothèque de films développée précédemment et améliorez la saisie de films en utilisant un deuxième formulaire.

Créez un nouveau projet puis, comme pour le projet précédent, ajoutez un contrôle de type *TreeView*, une *TextBox* et deux boutons. Ajoutez une *TextBox* avec un bouton **Ajout Catégorie**. Ajoutez un deuxième bouton que vous nommerez `Ajout Film`.

Ajoutez maintenant un nouveau formulaire au projet grâce à la méthode décrite précédemment. Dans le designer, sélectionnez l'onglet du nouveau formulaire et ajoutez-y deux boutons et une *TextBox*. À gauche de la *TextBox*, ajoutez un *Label* `Film`.

Sélectionnez le premier bouton, puis affichez les propriétés en appuyant sur F4. Dans la propriété *Text*, inscrivez `OK`. Changez le texte du deuxième bouton par `Annuler`. Il faut maintenant faire en sorte que, lorsque l'utilisateur clique sur le bouton OK, les informations du film s'ajoutent à l'arborescence.

Modifiez le code de l'événement "clic" du bouton **Ajout Catégorie** sur le formulaire 1. Recopiez le code suivant :

```
TreeView1.Nodes.Add(TextBox1.Text)
```

Cela permet de créer un nœud de catégorie. Modifiez ensuite le code de l'événement "clic" du bouton **Ajout Film**. Insérez les instructions suivantes :

```
Dim Form_ajout As Form2
Form_ajout = New Form2
Form_ajout.ShowDialog()
```

Il s'agit ici de déclarer une variable qui soit de type `Form2`, `Form2` étant le formulaire que vous avez ajouté à votre projet. De cette manière, un

clic sur le bouton **Ajout Film** ouvre une nouvelle fenêtre avec le contenu du formulaire 2.

Il faut maintenant modifier les événements "clic" du deuxième formulaire pour ajouter un film dans la catégorie sélectionnée.

Pour le bouton OK, saisissez le code suivant :

```
Form1.TreeView1.SelectedNode.Nodes.Add(TextBox1.Text)
Me.Dispose()
```

Pour le bouton **Annuler**, utilisez l'instruction suivante :

```
Me.Dispose()
```

De cette manière, un clic sur le bouton OK ajoute le film à la catégorie sélectionnée, et un clic sur le bouton **Annuler** ferme le formulaire. La méthode `Dispose` ferme le formulaire et efface toute trace de ce dernier dans la mémoire de l'ordinateur. Après l'exécution de la méthode `Dispose`, il est impossible d'accéder au contenu du formulaire.

Chapitre 6

Penser à l'utilisateur

Lorsque vous utilisez un programme, il est toujours plus agréable de se trouver face à une interface claire, où chaque bouton semble être à l’endroit où vous vous attendez à le trouver, où tous les menus portent des noms clairs et précis et qui vous apportent directement l’information que vous cherchez.

Maintenant que vous connaissez toutes les bases de la programmation, vous êtes capable de réaliser à peu près n’importe quel logiciel, aussi compliqué soit-il. C’est comme en cuisine : même la plus compliquée des pièces montées demande les mêmes ingrédients que le gâteau au yaourt. La suite de ce livre va vous apprendre à faire la différence entre un bon et un mauvais programme.

À la question "Qu’est-ce qu’un bon programme ?", n’importe quelle personne serait tentée de répondre "Un programme qui marche" et c’est une erreur. Fonctionner est la première chose à demander à un programme, la deuxième est de fournir un résultat en un temps minime. Le but de ce chapitre est de vous apprendre à vous placer du côté développeur et du côté utilisateur final. Dans le monde de l’entreprise, celui qui développe des logiciels de gestion pour différents clients et qui en fait le minimum pour les satisfaire au plus vite et développer davantage de projets fait une grande erreur. Que se passe-t-il lorsque le client n’est pas satisfait ?. Le développeur passe deux fois plus de temps à reprendre le projet et à le modifier qu’il ne l’aurait fait s’il s’était placé du côté utilisateur directement.

Pour bien faire, il faut réfléchir à trois points :

- l’interface principale ;
- les menus ;
- les valeurs entrées par l’utilisateur.

Le menu est la partie de l’application qui sert à réaliser les actions les plus courantes. Il s’agit en général d’une barre située juste sous la barre de titre de la fenêtre. Cette barre de menus doit être classique et le plus claire possible. Personne n’a envie de se perdre dans une hiérarchie de menus et de sous-menus pour trouver imprimer une carte de vœux. Pour cela, Visual Basic met à votre disposition des outils réellement pratiques.

L’interface, c’est le formulaire Windows. C’est ce que voit l’utilisateur lorsqu’il lance le programme. Il est donc de bon ton de la soigner. En effet, si le premier contact est réussi, 75 % du travail est fait, le

programme va plaire. Il faut pour cela réfléchir à l'organisation d'une part, qui doit rester logique et fonctionnelle, et au graphisme d'autre part, qui doit rendre l'application agréable à regarder. Pour arriver à un résultat beau et fonctionnel, vous allez apprendre à rendre un formulaire transparent, à l'organiser, et mettre en application un tas de petites astuces pour donner à une application un look professionnel.

Les retours des utilisateurs, c'est-à-dire ce qu'ils entrent dans le programme sous forme de valeurs, doivent faire l'objet d'une attention particulière de votre part. Imaginez qu'un utilisateur du programme d'annuaire veuille fournir l'âge d'une personne. Si au lieu d'entrer `32` dans la *TextBox* destinée à recueillir l'âge, il saisit `trente deux ans` ? Le programme va alors se fermer et l'utilisateur ne saura pas ce qu'il a fait de mal. Le deuxième problème relève de la sécurité. Comment garantir la sécurité des données d'un utilisateur ? Comment vérifier les saisies et déterminer si un utilisateur malveillant essaie de contourner la sécurité de l'application ? Toutes ces questions trouveront une réponse dans ce chapitre.

6.1. Les menus

Les menus représentent un moyen simple d'organiser l'accès à différentes ressources, mais comment les utiliser ?

La barre de menus classique

1 Lancez Visual Basic Express et créez un nouveau projet. Dans la boîte à outils, cherchez le contrôle nommé *MenuStrip* puis ajoutez-le au formulaire (voir Figure 6.1).

Notez la petite flèche blanche en haut à droite du contrôle *MenuStrip* : elle va permettre d'ajouter des éléments au menu.

2 Cliquez sur cette flèche : un menu s'affiche. Ignorez les options, à part celle qui propose *Insérez les éléments standard* (voir Figure 6.2).

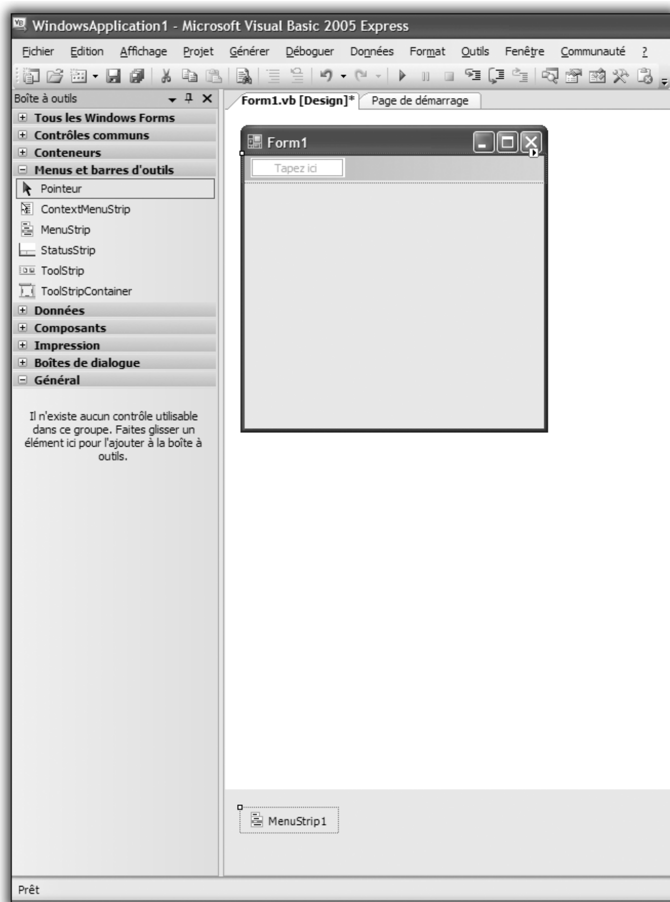


Figure 6.1 : *Ajout d’un menu*

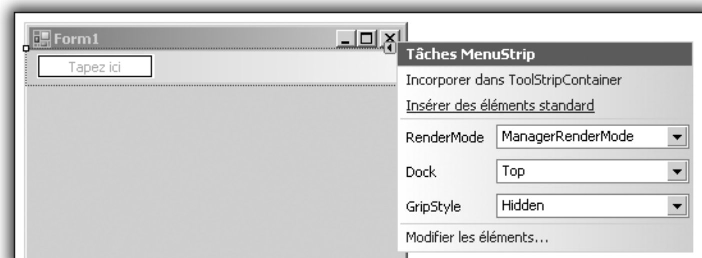


Figure 6.2 : *Insertion des éléments standard*

3 Cliquez dessus : la barre de menus contient tous les éléments courants d'une application Windows, à savoir les menus **Fichier**, **Edition**, **Outils** et **Aide**.

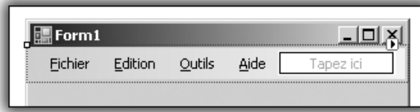


Figure 6.3 :
Menus standard

Pour le moment, vous avez tous les menus de premier niveau. Si vous cliquez sur chacun des éléments qui viennent d'être ajoutés, vous verrez qu'un sous-menu apparaît, vous laissant le choix d'autres éléments. Ainsi, le menu **Aide** a pour sous-menus **Index**, **Recherche**, **À propos**, qui sont les éléments de base d'un menu **Aide**. Faites le test. Lancez une application que vous utilisez tous les jours, comme votre logiciel de messagerie ou votre traitement de texte. Regardez la barre de menus. Elle contient les même éléments que ceux que vous venez d'ajouter. Compilez l'application avec la touche F5, puis cliquez sur l'un des boutons du menu : il ne se passe rien. C'est normal : pour l'instant, vous avez simplement ajouté un squelette de menus à l'application. Maintenant, vous allez lui ajouter des muscles pour qu'elle puisse être pleinement fonctionnelle.

4 Revenez dans l'éditeur de formulaire, puis cliquez à droite du menu **Help**. Vous pouvez ajouter votre propre menu. Dans la zone de saisie, entrez `Texte`. Vous allez ajouter un menu permettant le contrôle du texte d'une *TextBox*.

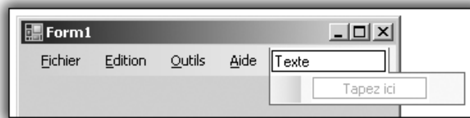


Figure 6.4 :
Ajout d'un élément de menu

Pour cela, vous allez avoir besoin d'une *TextBox*. Ajoutez au formulaire un contrôle *RichTextBox* en la faisant glisser depuis la boîte à outils.

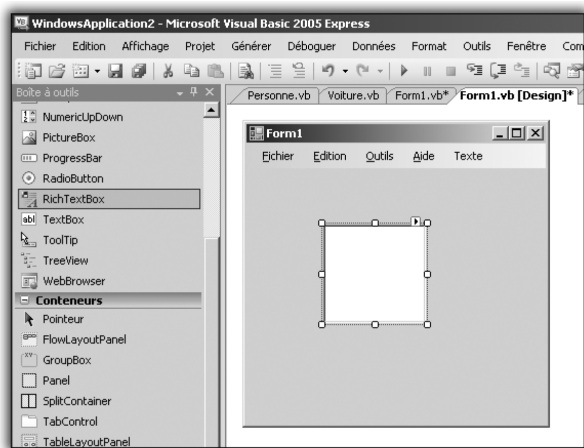


Figure 6.5 : *Ajout d'une RichTextBox*



RichTextBox

C'est une *TextBox* enrichie d'un lot de fonctionnalités utiles, par exemple la possibilité d'effectuer un traitement sur une sélection de texte.

5 Une fois arrivé à ce résultat, cliquez sur le menu **Texte** que vous avez inséré, puis ajoutez-lui un sous-menu en remplissant la case située juste dessous. Appelez ce sous-menu `Gras`.

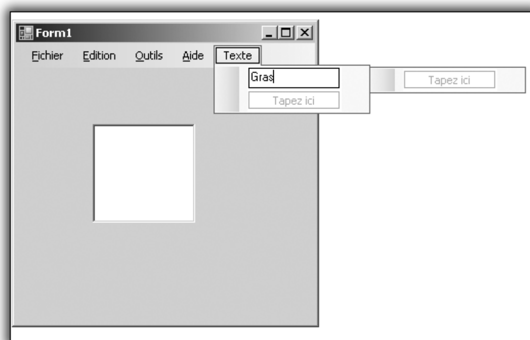


Figure 6.6 : *Ajout d'un sous-menu*

Maintenant, ajoutez une action à ce sous-menu.

6 Double-cliquez sur le sous-menu que vous avez créé, ce qui va vous transporter dans le code au niveau de l'événement "clic sur le sous-menu". À cet endroit, ajoutez cette ligne de code :

```
RichTextBox1.SelectionFont = New Font(RichTextBox1
✂ .SelectionFont, FontStyle.Bold)
```

Ce code agit directement sur le texte sélectionné dans la *RichTextBox* et lui applique un style de police gras.

7 Lancez l'application en appuyant sur la touche F5, puis entrez du texte dans la *RichTextBox*. Ensuite, sélectionnez-en une partie et cliquez sur le menu **Texte** puis sur le sous-menu **Gras**.



Figure 6.7 :
Le texte en gras

Le texte sélectionné vient de passer en gras, mais pas le reste du texte. Pour vérifier que vous avez bien saisi le concept de menu, ajoutez des sous-menus au menu **Texte** pour donner un style italique ou souligné.

Les barres d'outils

De nombreuses applications proposent une barre d'outils généralement située sous la barre de menus. Une barre d'outils propose un raccourci vers une fonction souvent utilisée, ce qui évite à l'utilisateur de chercher dans l'architecture des menus.

Vous allez reprendre l'exemple précédent avec la *RichTextBox* et ajoutez quelques raccourcis de fonction.

Partez de ceci :



Figure 6.8 : *Ajout des outils*

Vous allez ajouter un contrôle *ToolStrip* depuis la boîte à outils.

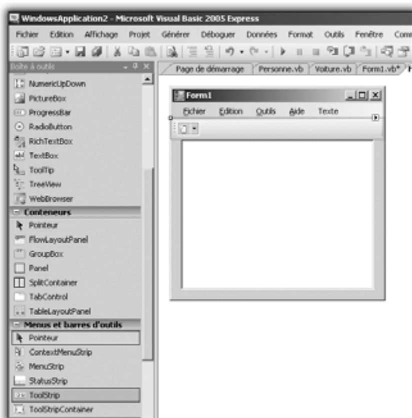


Figure 6.9 : *Ajout d'un ToolStrip*



ToolStrip

C'est une boîte à outils qui propose plusieurs options permettant de gérer les icônes de la barre d'outils ainsi que sa position.

1 Cliquez sur la petite flèche blanche en haut à droite du *ToolStrip* puis sur *Insérer les éléments standard*.

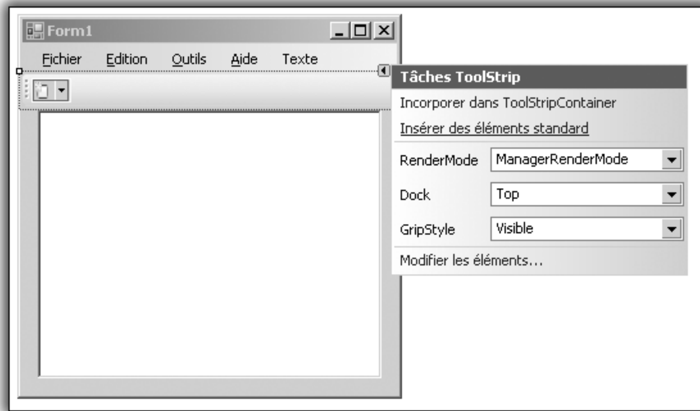


Figure 6.10 : *Insertion des outils standard*

Comme précédemment, cette action insère les éléments les plus couramment utilisés : *Ouvrir un fichier*, *Sauvegarder un fichier*, etc.

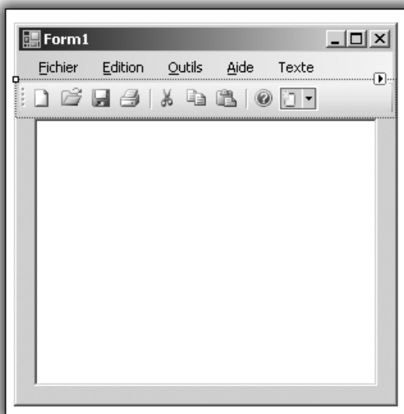


Figure 6.11 : *ToolStrip standard*

Il existe un deuxième moyen qui permet plus de contrôle sur la barre d'outils. Une fois le *ToolStrip* ajouté, dans la fenêtre des propriétés, une collection *Items* permet d'éditer les boutons de la barre.

2 Cliquez sur *Collection* puis sur l'icône décorée de trois points de suspension juste à côté.

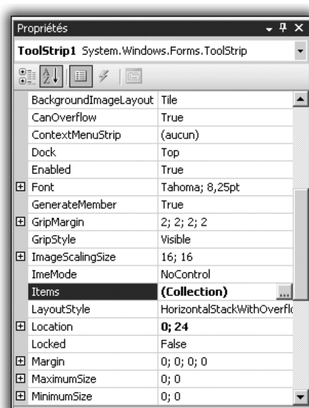


Figure 6.12 :
Lancer l'éditeur de ToolStrip

Cela a pour effet de lancer l'éditeur de *ToolStrip*.

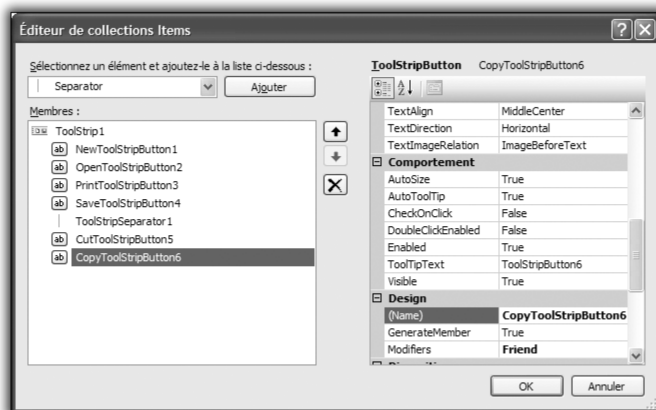


Figure 6.13 : *Éditeur de ToolStrip*

Dans cet éditeur figurent à droite les boutons déjà ajoutés par Visual Basic après le clic sur *Insérer les éléments standard*. Pour ajouter un élément, il faut utiliser le bouton **Ajouter** en haut à gauche de l'éditeur. À gauche de ce bouton se trouve une liste de choix permettant de

sélectionner l'élément à ajouter à l'application. Pour le moment, ajoutez un bouton, que vous renommerez par G.

1 Dans l'éditeur, sélectionnez le bouton que vous venez d'ajouter, puis cherchez dans la liste des propriétés à droite, la propriété appelée *DisplayStyle* et changez sa valeur en *Text*.

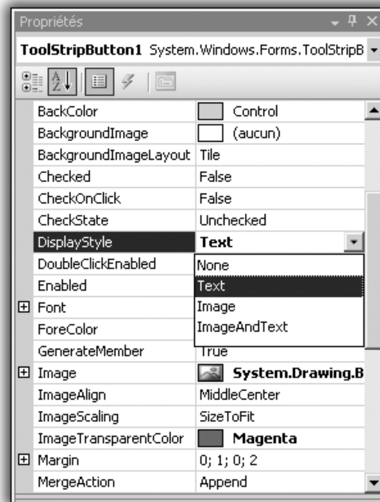


Figure 6.14 : *Changez le DisplayStyle*

2 Ensuite, cherchez la propriété *Text* et changez la valeur en G. Vous avez maintenant un bouton qui dispose d'un style réellement professionnel.

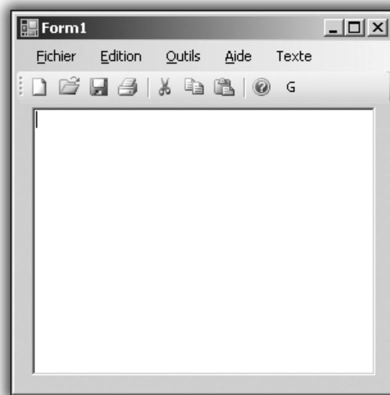


Figure 6.15 : *Ajout d'un élément de ToolStrip*

Ajoutez deux autres boutons pour les styles italique et souligné.

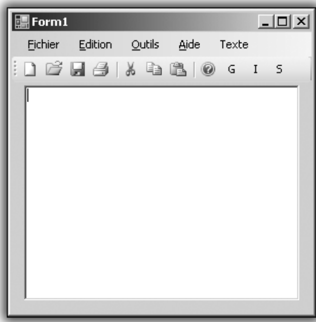


Figure 6.16 :
Outils de mise en forme

3 Revenez maintenant dans le designer, puis double-cliquez sur les boutons pour ajouter du code au niveau de leur événement "clic".

Pour le bouton "gras", ajoutez la ligne suivante :

```
RichTextBox1.SelectionFont = New Font(RichTextBox1
✂ .SelectionFont, FontStyle.Bold)
```

Pour le bouton "italique", ajoutez la ligne suivante :

```
RichTextBox1.SelectionFont = New Font(RichTextBox1
✂ .SelectionFont, FontStyle.Italic)
```

Pour le bouton "souligné", ajoutez la ligne suivante :

```
RichTextBox1.SelectionFont = New Font(RichTextBox1
✂ .SelectionFont, FontStyle.Underline)
```

4 Lancez le programme en appuyant sur la touche F5, saisissez du texte dans la *RichTextBox*, puis sélectionnez une partie du texte. Utilisez l'un des boutons et admirez le résultat.



Figure 6.17 :
Mini-éditeur de texte

Vous avez maintenant une petite application de traitement de texte qui permet de modifier le style du texte saisi.

Vous allez modifier les boutons standard pour qu'ils ouvrent un document.

1 Double-cliquez sur l'icône de l'ouverture de fichier dans le *ToolStrip* après avoir ajouté des éléments standard. Vous devriez être dans la partie d'édition du code. Ajoutez les lignes suivantes :

```
If (OpenFileDialog1.ShowDialog() = Windows.Forms
✂ .DialogResult.OK) Then
RichTextBox1.Text = _
System.IO.File.ReadAllText(OpenFileDialog1.FileName)
End If
```



Les sauts de ligne

Dans Visual Basic Express, vous ne pouvez pas entrer d'instructions sur plusieurs lignes à moins de terminer la ligne à prolonger par un caractère de soulignement. Ainsi vous pouvez manipuler plus facilement de longues instructions.

La première ligne permet de tester si l'utilisateur a bien validé avec le bouton OK dans la boîte de dialogue de sélection de fichier. La deuxième ligne permet de récupérer le contenu textuel du fichier choisi dans la boîte de dialogue (`OpenFileDialiog1.Filename`).

Petit problème : cette méthode ne permet de lire qu'un document de texte, enregistré avec le Bloc-notes de Windows. Pour charger un document de type Richt Text Format, qui est un format beaucoup plus standard, vous pouvez utiliser la méthode `LoadFile` de la *RichTextBox*.

2 Remplacez la deuxième ligne par :

```
RichTextBox1.LoadFile(OpenFileDialog1.FileName)
```

Cela chargera directement un fichier dans la zone d'édition de texte de la *RichTextBox*. Autre avantage de cette méthode : elle permet de garder le formatage du texte. Par exemple, si dans un éditeur de type Wordpad ou Microsoft Word, vous avez du texte en gras, le texte en gras sera chargé par la *RichTextBox*.

Maintenant que vous avez un moyen d'ouvrir les documents au format RTF, vous allez implémenter une fonction de sauvegarde de ces documents. Ajoutez au formulaire un contrôle de type *SaveFileDialog*.

3 Double-cliquez sur l'icône en forme de disquette du `ToolStrip` de manière à vous placer dans l'édition de code de l'événement "clic" du bouton et copiez les lignes suivantes :

```
SaveFileDialog1.Filter = "Fichiers RTF(*.rtf)|.rtf"
If (SaveFileDialog1.ShowDialog() = Windows.Forms
✂ .DialogResult.OK)_
 Then
RichTextBox1.SaveFile(SaveFileDialog1.FileName)
End If
```

La première ligne permet de spécifier le filtre à appliquer pour trouver les fichiers à sauvegarder. Dans ce cas, il s'agit des fichiers RTF. Vous appliquez donc un filtre de type **.rtf*. Sur la deuxième ligne, vous vérifiez que l'utilisateur a bien cliqué sur le bouton OK de la boîte de dialogue. S'il choisit le bouton **Annuler**, vous n'effectuez pas la sauvegarde. Via la troisième ligne, vous allez effectuer la sauvegarde. Pour cela, vous utilisez la méthode `SaveFile` de la *RichTextBox*, qui prend en paramètre un chemin vers un fichier, qui est ici donné par le résultat de la boîte de dialogue **SaveFileDialog** ajoutée précédemment.

Vous avez maintenant un petit éditeur de texte, qui permet l'édition, l'ouverture et la sauvegarde de fichiers. Vous allez ajouter un système de copier/coller. Parmi les boutons standard du *ToolStrip* se trouvent les trois boutons classiques de couper, copier et coller.

4 Placez-vous dans l'éditeur de code du bouton **Copier** et insérez le code suivant :

```
RichTextBox1.Copy()
```

Cela aura pour effet de copier le texte sélectionné dans le Presse-papiers Windows.

5 Placez-vous maintenant dans le code de l'événement "clic" du bouton **Coller** pour ajouter la ligne suivante :

```
RichTextBox1.Paste()
```

Cela aura pour effet de copier le contenu du Presse-papiers là où se trouve le curseur dans la *RichTextBox*. Attention toutefois, si vous

cliquez sur **Coller** alors que du texte est sélectionné, seul le texte sélectionné sera remplacé.

Pour terminer la petite application, ajoutez une fonction **Couper** pour copier un texte sélectionné et le supprimer de la *RichTextBox*.

6 Double-cliquez sur le bouton **Couper** du *ToolStrip* dans le designer, puis une fois dans le code, ajoutez la ligne suivante :

```
RichTextBox1.Cut()
```

Cette instruction va couper le texte, c'est-à-dire le supprimer de la *RichTextBox* et placer le contenu de la sélection dans le Presse-papiers en vue d'une réutilisation ultérieure.

Lancez maintenant le programme en appuyant sur la touche F5 et testez les nouvelles fonctionnalités.

Il reste maintenant trois boutons auxquels vous n'avez pas ajouté de fonctionnalités. **Nouveau** est le premier des boutons standard du *ToolStrip*. Il permet normalement d'ouvrir un nouveau document du type par défaut de l'application. Dans cet exemple, vous allez vous contenter de vider le contenu de la *RichTextBox* pour repartir sur un document "frais".

7 Double-cliquez sur le bouton **Nouveau** dans le designer de formulaire puis ajoutez le code suivant une fois dans l'éditeur.

```
RichTextBox1.Clear()
```

Maintenant, lorsque vous cliquez sur l'icône *Nouveau*, vous videz la *RichTextBox*. Il reste les boutons **Imprimer** et **Aide**. L'impression n'étant pas possible directement à partir de la *RichTextBox*, vous allez supprimer le bouton. Pour supprimer un bouton d'un *ToolStrip*, vous avez deux solutions. La première consiste à sélectionner le bouton en cliquant dessus dans le designer puis à appuyer sur la touche Suppr. La deuxième consiste à lancer l'éditeur de collection de l'objet *ToolStrip*, à chercher le bouton qui correspond au bouton **Imprimer** dans la liste des éléments du *ToolStrip* et à cliquer sur l'icône représentant une croix à droite de la liste des éléments.

Il reste le bouton d'aide. Pour sa gestion, vous allez charger une aide au format HTML (le format classique des aides Windows et des sites Internet). Vous pourrez au choix charger un fichier local ou une adresse

Internet. Double-cliquez sur le bouton **Aide** ajouté au *ToolStrip* et ajoutez la ligne suivante dans l'éditeur :

```
Help.ShowHelp(ToolStrip1, "c:\help.htm")
```

Cette instruction va charger un fichier HTML qui se situe à la racine du disque dur principal et ouvrir un navigateur qui pointera directement sur cette page.

Si vous ne maîtrisez pas HTML, voici un exemple que vous pouvez recopier dans un document texte :

```
<HTML>
<HEAD>
<TITLE>Page d'aide de MonApplication<TITLE>
</HEAD>
<BODY>
Pour le moment, aucune aide n'est disponible.
</BODY>
</HTML>
```

Enregistrez ensuite le document à la racine du disque dur principal et nommez-le `help.htm`.



Figure 6.18 :
Aide de l'application

Maintenant l'application d'édition de texte est complète. Il ne reste plus qu'à éditer les différents menus pour reprendre les fonctionnalités qui y sont décrites et supprimer celles qui ne sont pas implémentées.

À présent, les menus n'ont plus de secret pour vous. Voyons maintenant les éléments qui donnent aux applications un look professionnel et les

erreurs à ne pas commettre pour ne pas noyer l'utilisateur dans une interface inutilisable au quotidien.

6.2. Créer un lien homme-machine

Dans le domaine de la programmation, il est courant de parler d'interface homme-machine, ou IHM. Ce terme est une sorte de fourre-tout dont on se sert pour évoquer ce qui assure une interactivité avec l'utilisateur. Malheureusement, il n'existe pas de règle stricte permettant d'arriver à une interface universellement parfaite. Qu'est-ce qui fait une bonne IHM ? C'est une question à laquelle il convient de répondre en trois points.

Être clair

Cette première directive est double, il s'agit d'une part de ne pas surcharger le visuel de l'utilisateur, par exemple de ne pas multiplier les boutons ou encore de privilégier les titres courts tel que "Imprimer" plutôt que "Lancer l'impression de la feuille en cours". Vous venez de voir comment utiliser les menus, donnez-leur des noms clairs et concis.

Toute la justesse d'une bonne interface tient en l'équilibre entre les fonctions visibles et les fonctions qui le sont moins. Si vous développez un programme destiné à un environnement sous Windows, la plupart des utilisateurs sont déjà "formatés" par les logiciels qu'ils emploient tous les jours. Leur premier réflexe lorsqu'ils chercheront une fonctionnalité sera d'aller voir au même endroit que dans leur programme habituel. C'est pourquoi dans la plupart des applications Windows, la commande **Quitter** est située dans le menu **Fichier**. C'est une sorte de convention entendue, rien n'est écrit nulle part, mais vous risquez de déboussoler l'utilisateur si vous ne la respectez pas.

Organiser

Le deuxième conseil à suivre est de garder une bonne organisation, de préférence par thèmes. Que viendrait faire un bouton **Imprimer** dans un groupe de boutons destinés aux modes de visualisation d'une image ? Rien. Pour cela, Visual Basic met à votre disposition plusieurs niveaux de menus, d'icônes de lancement rapide via les *ToolStrip*. Il existe

également d’autres manières d’harmoniser l’interface de vos programmes que vous allez voir ci-après.

N’hésitez pas à multiplier les formulaires Windows. Dans l’idée de regrouper les éléments d’une même famille, s’il faut par exemple saisir des informations, mieux vaut le faire dans une fenêtre séparée de la fenêtre principale. Pour la saisie des informations d’une personne (un nom, un prénom, une adresse e-mail et un numéro de téléphone), mieux vaut utiliser un deuxième formulaire et le relier à un bouton **Ajouter une entrée** sur le premier formulaire. N’oubliez jamais qu’un projet peut contenir plusieurs formulaires et qu’ils peuvent être reliés entre eux de manière simple. Vous verrez comment procéder dans l’exemple suivant.

Faire du beau

Il convient en matière de graphisme d’aborder l’idée de "belle interface". Lors de l’élaboration d’un programme, restez dans les limites du développement professionnel, traditionnellement beaucoup de gris, très peu de fantaisie, etc. Vous pouvez cependant essayer de faire la différence avec les autres programmes gris à l’aide d’effets de transparence, jouer sur l’opacité, les formes, la couleur, sans toutefois tomber dans l’excès d’une application à fond rose aves des boutons verts. La fenêtre de propriétés des contrôles Windows propose un lot de propriétés destinées à ces effets. Vous allez les mettre à profit dès maintenant.

La transparence

Un bon moyen de surprendre les utilisateurs est de jouer sur la forme de l’application. Pour cela, ne changez pas directement la forme du formulaire dans le designer. Rusez. Vous allez en fait déclarer au formulaire qu’une couleur est utilisée comme couleur de transparence puis afficher une image en arrière-plan pour rendre le formulaire transparent. Ouvrez un éditeur d’image, comme Microsoft Paint et appliquez quelques touches de bleu primaire au formulaire.

Si vous êtes en manque d’inspiration, vous pouvez reprendre ce modèle :

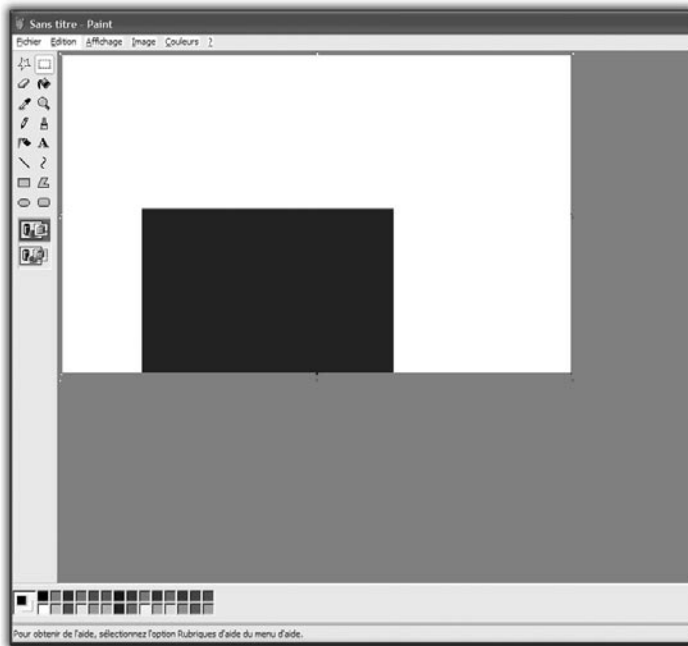


Figure 6.19 : *Le futur fond de l'application*

Une fois que vous avez fini de modeler l'image de fond, enregistrez-la au format Bitmap, (extension *.bmp*).

1 Créez un nouveau projet dans Visual Basic et sélectionnez le formulaire dans le designer en cliquant dessus. Dans la fenêtre des propriétés, cherchez la propriété *BackgroundImage*. Cliquez sur la case à droite de celle-ci puis sur **Importer** dans le gestionnaire d'images.

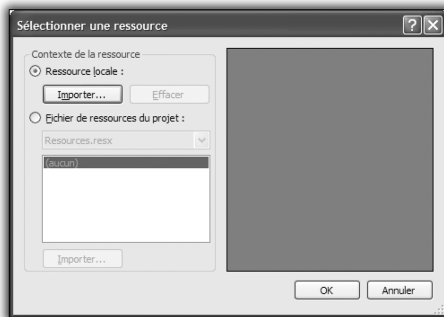


Figure 6.20 : *Gestionnaire d'images*

2 Sélectionnez l'image que vous venez d'enregistrer puis cliquez sur OK.

Normalement l'image s'affiche sur la forme, mais peut-être pas correctement du fait de sa taille trop élevée. Juste en dessous de la propriété *BackgroundImage*, se situe la propriété *BackgroundImageLayout*. Elle permet de modifier la manière dont l'image est affichée sur le formulaire. Changez la valeur en cliquant sur la case d'édition de la propriété, puis sélectionnez *Stretch* ou *Étirer*. L'image s'adapte au formulaire. Il reste à rendre le carré bleu transparent.

3 Pour cela, descendez dans la fenêtre des propriétés et cherchez la propriété *TransparencyKey*.



TransparencyKey

La clé de transparence est un code qui permet de définir sur le formulaire une couleur qui ne s'affichera pas, ou plutôt qui sera complètement transparente.

Modifiez sa valeur pour obtenir un bleu primaire utilisé pour l'image de fond. Pour cela, cliquez et modifiez le texte par `Bleu`. Lancez maintenant le programme en appuyant sur la touche F5. Déplacez la fenêtre un peu partout sur votre écran. Vous pouvez voir à travers le formulaire à l'endroit où il est bleu.

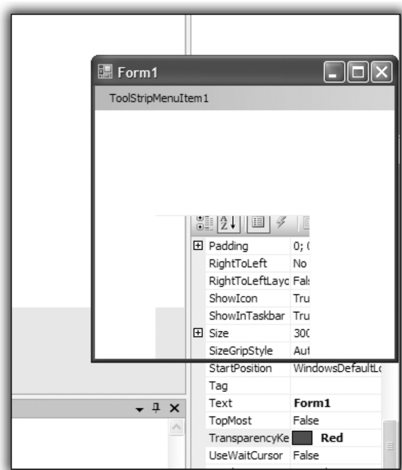


Figure 6.21 : *Une partie de la fenêtre est transparente*

L'opacité

La transparence pose ici un problème. En effet, il n'existe aucune graduation. Ou bien une partie du formulaire est transparent, ou bien elle ne l'est pas. Pour avoir une graduation, il faut recourir à la propriété *Opacity* du formulaire. Dans cet exemple, vous allez ajouter un contrôle *TrackBar* pour modifier l'opacité du formulaire principal. Avec la propriété *Opacity*, l'intégralité du formulaire devient plus transparent. Dans l'exemple précédent, si un bouton avait été placé sur le formulaire à l'endroit où ce dernier est transparent, il aurait été visible pleinement sans aucun effet. Avec une modification de l'opacité, tous les contrôles deviennent moins opaques.

1 Créez un nouveau projet dans Visual Studio.
2 Cliquez sur le formulaire. Dans la boîte des propriétés, cherchez la valeur *Opacity* à droite de la propriété.
3 Modifiez la valeur en inscrivant `50` dans la case prévue.

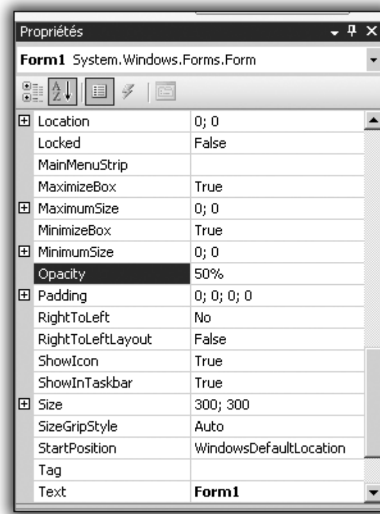


Figure 6.22 :
Modification de l'option Opacity

Cela aura pour effet de réduire de moitié la transparence de la fenêtre. Lancez le programme en appuyant sur la touche F5 et admirez le résultat.

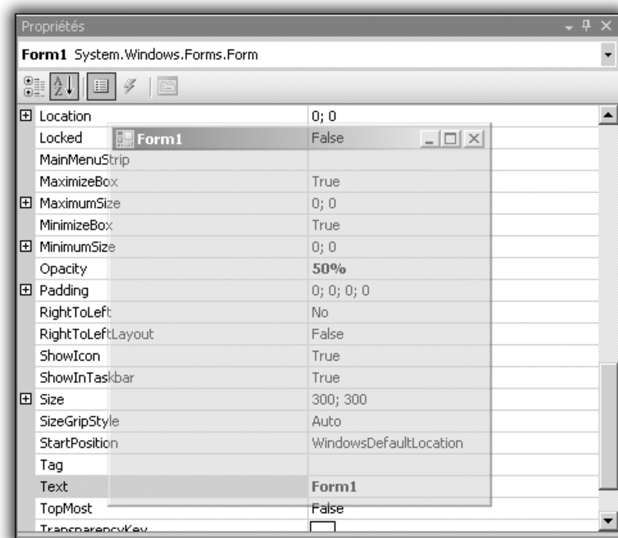


Figure 6.23 : *Une fenêtre à moitié transparente*

Vous allez maintenant graduer cette opacité. Revenez au designer en fermant le test de l’application.

4 Ajoutez un contrôle *TrackBar* via la barre d’outils de Visual Basic Express.

La *TrackBar* est une barre de recherche qui permet de récupérer une valeur dans un intervalle donné. Une fois qu’elle est ajoutée au formulaire, allez dans la fenêtre des propriétés de la *TrackBar* et cherchez la propriété *Minimum*. Il s’agit de la borne inférieure de l’intervalle représenté par la barre. Inscrivez la valeur `1`. Cette valeur va représenter la valeur minimale de visibilité de la fenêtre. Une valeur de 0 ferait disparaître totalement le formulaire et vous ne pourriez plus cliquer dessus.

De la même manière, cherchez la propriété *Maximum* de la *TrackBar* et changez sa valeur en `10`. C’est la valeur maximale que peut atteindre l’opacité de la fenêtre. 100 % rend la fenêtre complètement opaque.

Lancez le programme en appuyant sur la touche F5. Cliquez sur le curseur de la *Trackbar* puis faites-le glisser de gauche à droite. Il ne se passe rien car à aucun moment vous n’avez fait le lien entre la valeur du curseur sur la barre et l’opacité de la fenêtre. Remédiez à cela. Revenez

au designer de Visual Studio et cliquez sur la *TrackBar*. Dans la fenêtre des propriétés, affichez les événements en cliquant sur l'icône en forme d'éclair, puis dans la liste, cherchez l'événement nommé *ValueChanged*. Double-cliquez sur la case à droite du nom, ce qui vous place dans l'éditeur de code, à l'endroit des instructions effectuées lors de cet événement. Recopiez cette ligne :

```
Me.Opacity = TrackBar1.Value/10
```

Lancez ensuite le programme en appuyant sur F5. Bougez maintenant le curseur de la *TrackBar* : l'opacité de la fenêtre est modifiée en temps réel.

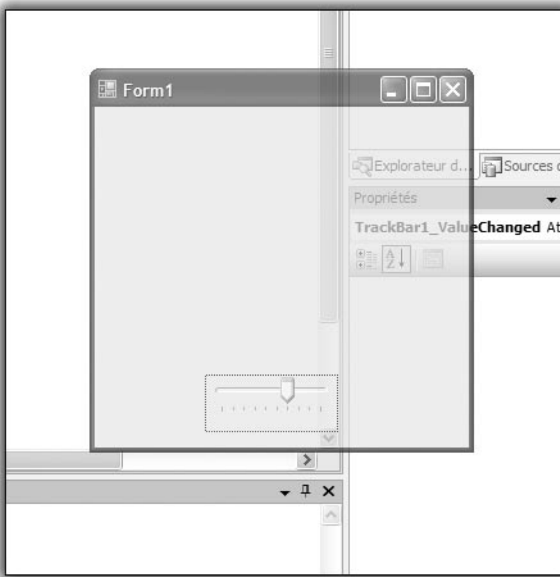


Figure 6.24 : *Modification dynamique de l'opacité*

La ligne de code entrée dans l'événement est facile à comprendre : chaque fois que la valeur de l'opacité de la *TrackBar* change, elle est enregistrée dans la variable définissant l'opacité de la fenêtre et de ses contrôles. Elle est divisée par 10 car elle doit être comprise en 0 et 1, la propriété *Opacity* attendant une variable de type Double.

L'ancrage, le docking et le regroupement

Jusqu'à maintenant, tous les formulaires que vous avez dessinés posent un problème de redimensionnement. Lorsque vous lancez le programme,

la fenêtre principale s'affiche à la taille que vous lui avez donnée dans le designer. Mais que se passe-t-il si vous la redimensionnez pour qu'elle occupe tout l'écran ? Faites le test. Dans le designer de formulaire de Visual Studio, ajoutez plusieurs contrôles de type bouton, une *PictureBox*, et lancez-le. Cliquez sur le bouton d'agrandissement de la fenêtre. Que se passe-t-il ? Les contrôles ne se repositionnent pas directement sur le formulaire et l'application paraît "désordonnée". Pour remédier à cela, il faut avoir recours à l'ancrage des contrôles. L'ancrage revient à préciser de manière relative à quels endroits doivent s'accrocher les contrôles dans le dessin. Pour ce faire, il faut découper le formulaire en zones. Il existe un moyen simple de procéder. Vous allez utiliser le contrôle *TableLayout*. Faites glisser un *TableLayout* sur une forme dans un nouveau projet. Le *TableLayout* est en fait une grille dans laquelle vous allez pouvoir placer les différents contrôles. Quand la fenêtre est redimensionnée, la grille l'est aussi et les contrôles gardent leur place. Par défaut, la grille est divisée en quatre, deux lignes et deux colonnes. Vous pouvez éditer le nombre de lignes et de colonnes en cliquant sur le contrôle puis sur la petite flèche blanche en haut à droite. S'affichent alors des options d'édition de lignes et de colonnes.

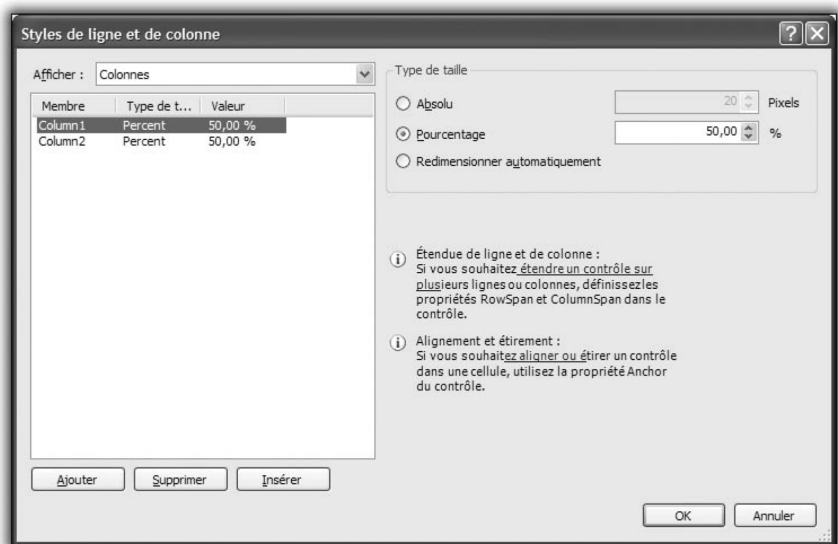


Figure 6.25 : *L'éditeur de grille*

Il reste maintenant à travailler sur le positionnement des contrôles dans les différentes parties de la grille. Ajoutez un bouton dans l'une des parties. Le bouton va par défaut se caler en haut à gauche de la grille.

Cela est directement lié à la propriété *Anchor* du bouton. Cliquez sur la case à droite de la propriété *Anchor*.

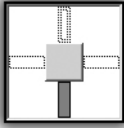


Figure 6.26 :
Propriété Anchor

Vous pouvez indiquer par rapport à quelle partie du contrôle parent doit être placé un contrôle. Si vous grisez la case haute, le contrôle sera placé au milieu et en haut. Si vous grisez le bas, le contrôle sera placé en bas. Si le haut et le bas sont grisés, le contrôle sera centré. Testez sur le bouton. Modifiez la propriété *Anchor* et lancez le programme pour observer les changements.



Anchor

La propriété *Anchor* permet de définir les bords du formulaire par rapport auxquels le contrôle doit être centré.

Ajoutez une *PictureBox* dans l'une des cases de la grille. Dans la fenêtre des propriétés, cherchez la propriété *Dock* et cliquez sur la case de droite.



Figure 6.27 :
Le choix du Dock

Les différents choix possibles permettent pour le haut, le bas, la droite et la gauche de bloquer l'alignement du contrôle. Si vous choisissez de le centrer, le contrôle va prendre toute la place disponible du contrôle parent. Dans le cas de la *PictureBox*, il s'agit de la case du *TableLayout* dans lequel vous l'avez placée. Si vous ne mettez pas de *TableLayout*, le contrôle parent se trouve être le formulaire lui-même. Donc si vous choisissez de le centrer, le contrôle va remplir tout le formulaire.



Docking

Le docking permet de garder la relativité de la position, par exemple le fait d'être collé au bord droit même si le formulaire est redimensionné.

Les derniers éléments qui peuvent servir pour redimensionner, garder les proportions, grouper les contrôles sont les contrôles *GroupBox* et *Panel*.



GroupBox

Le contrôle *GroupBox* permet de regrouper des contrôles, tout comme le contrôle *Panel*, à la différence que, pour la *GroupBox*, les contrôles sont entourés d'un liseré fin et d'une légende. En outre, lorsque vous regroupez des choix à l'aide de contrôle de type *RadioButton*, ils peuvent être sélectionnés de manière exclusive.

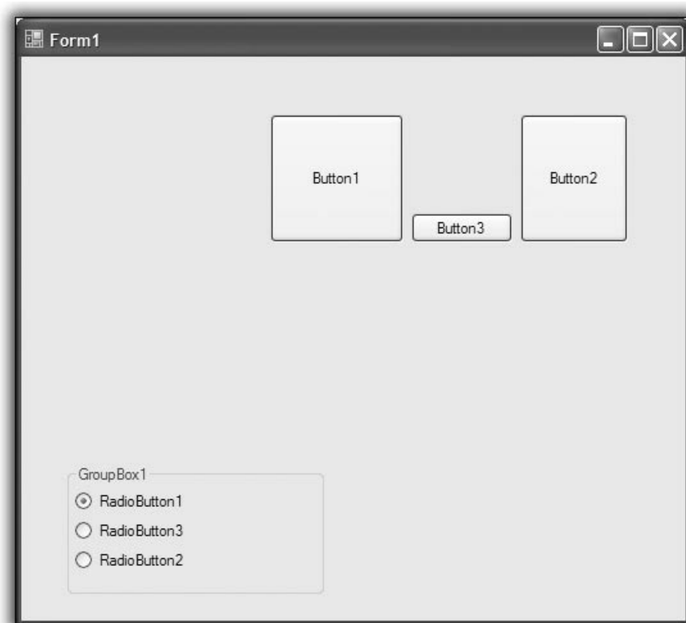


Figure 6.28 : *Une GroupBox et un Panel*

Dans cet exemple, il suffit d'ajouter à un formulaire un *Panel* et une *GroupBox*. Pour le *Panel*, il faut modifier la propriété *Anchor* à haut et droite, et pour la *GroupBox* à bas et gauche. Pour le *Panel*, les propriétés

Dock des boutons ont été modifiées pour droite sur le bouton 2, gauche pour le bouton 1 et bas pour le bouton 3. Remarquez comment ils se placent dans le *Panel*.

Reproduisez le même exemple dans un nouveau projet. En lançant le programme, remarquez que les *RadioButton* sont indépendants les un des autres. Si vous ajoutez une deuxième *GroupBox*, vous pouvez réaliser un deuxième groupe de *RadioButton* indépendants, ce qui est impossible s'ils ne sont pas regroupés.

6.3. Attention aux pirates !

Lors de l'utilisation du programme, l'utilisateur aura probablement à entrer des valeurs, par exemple ses informations personnelles. Il est important de faire attention à la façon de stocker ces informations, car les rendre facilement accessibles à n'importe qui peut entraîner un lot de complications vraiment fâcheuses. Imaginez par exemple que lors de l'utilisation d'un logiciel permettant le paiement en ligne de la licence, le numéro de carte bleue de l'utilisateur puisse être vu par tous ceux qui utilisent le programme après lui. Ce programme ne risque pas de rester longtemps installer sur la machine de l'utilisateur. Évitez de stocker des informations relatives à l'utilisateur de manière claire.



Vous apprendrez au chapitre ***Enregistrer des données*** *à stocker les informations fournies par les utilisateurs.*

Pour le moment, voici quelques recommandations à propose des manipulations d'informations saisies par les utilisateurs. On distingue les utilisateurs normaux, qui essayent simplement d'exploiter le programme mis à leur disposition, et les utilisateurs malveillants, qui veulent tirer profit du programme.

Penser aux vérifications

Il est important de prendre en compte le point de vue de l'utilisateur. Ne partez jamais du principe que quelqu'un utilisant pour la première fois le programme parviendra sans problème à s'en servir. Prenons le cas d'un formulaire utilisateur qui demande l'âge d'une personne. Comment savoir s'il faut entrer 32, 32 ans ou trente-deux ans ? Pour aider

l'utilisateur, il faut placer une information sur l'interface, ou même réduire les possibilités de saisie à un ensemble de valeurs.

Pour comprendre cela, prenez un nouveau projet. Sur le formulaire principal, ajoutez une *TextBox* et un contrôle *Button*.

Dans le code de l'événement "clic" du bouton, ajoutez le code suivant :

```
MessageBox.Show(Int32.Parse(TextBox1.Text).ToString)
```

Lancez maintenant l'application, entrez ce que vous voulez dans la *TextBox* et cliquez sur le bouton. Vous vous retrouvez alors face à une erreur.

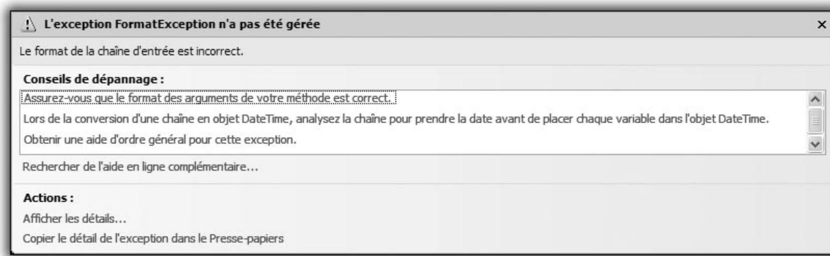


Figure 6.29 : *Erreur de format*

Lorsque le code essaie de transformer le contenu de la *TextBox* en texte, il se retrouve bloqué par le fait que le texte n'est pas sous forme numérique. Le programme s'arrête donc de fonctionner. Pensez donc à être le plus explicite possible quant aux saisies des utilisateurs.

Il existe toujours un jeu du chat et de la souris entre le développeur d'une application et ceux qui l'utilisent de façon malveillante. À chaque fois qu'un développeur protège une information, quelqu'un va chercher à découvrir cette information.

Pour l'instant, vous allez vous contenter de restreindre l'utilisation de l'application aux personnes disposant d'un mot de passe.

1 Dans l'Explorateur de solutions, cliquez du bouton droit sur le nom du projet puis sur **Ajouter**, et enfin sur **Nouvel élément**.
2 Dans les choix qui s'offrent à vous, sélectionnez **LoginForm**.

Il s'agit tout simplement d'une fenêtre de type formulaire qui contient déjà tous les éléments nécessaires pour identifier un utilisateur.

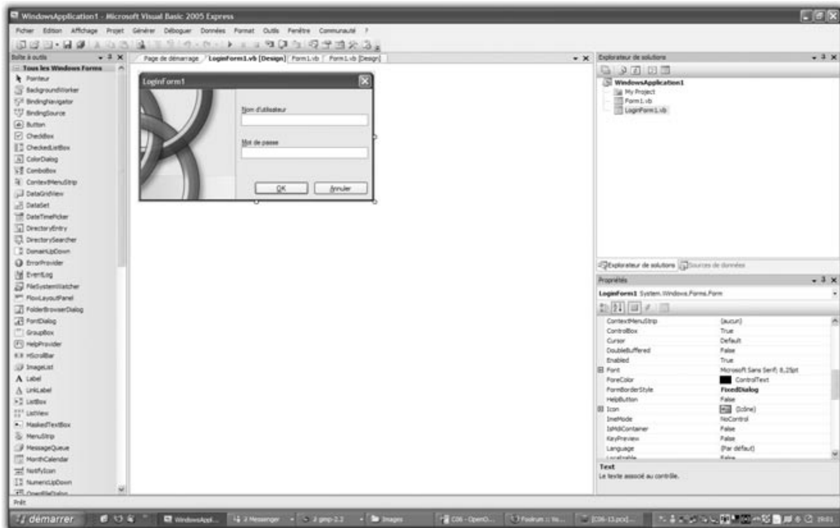


Figure 6.30 : *Le nouveau LoginForm*

En double-cliquant sur le bouton OK du *LoginForm* dans le designer, vous allez pouvoir modifier le code exécuté lors d'un clic sur le bouton OK du *LoginForm*.

3 Modifiez le code de l'événement "clic" du bouton en recopiant ceci :

```
If Me.PasswordTextBox.Text = "cafetiere" Then
  Me.Close()
End If
```

Cela précise au *LoginForm* que, si le texte entré dans le champ *Password* est `cafetiere`, l'utilisateur a donné le bon mot de passe et est alors autorisé à utiliser le programme.

Maintenant il faut que vous utilisiez ce *LoginForm* au moment où le programme se charge pour que l'utilisateur ne puisse pas utiliser le programme avant de s'être authentifié. Pour cela, dans le designer, double-cliquez sur la fenêtre principale du programme.

4 Entrez la ligne suivante dans l'éditeur de code :

```
LoginForm1.ShowDialog()
```

De cette manière, avant que la fenêtre principale du programme n'apparaisse, le formulaire d'identification sera chargé en premier et ne chargera le formulaire principal que si le mot de passe est le bon.

Il reste encore un petit problème à régler. En effet, si vous cliquez sur le bouton **Cancel** du *LoginForm*, vous avez accès au formulaire principal même si le mot de passe n'est pas le bon. Auriez-vous pensé à vérifier le comportement du bouton **Annuler** ? Cela fait partie des tests de sécurité à faire impérativement lors de tout développement. Il faut toujours être sûr de la façon dont va réagir l'application lorsque l'utilisateur fera tout et n'importe quoi.

5 Modifiez le code du bouton **Annuler** et remplacez-le par :

```
Application.Exit
```

Cette ligne terminera l'exécution du programme.

Vous avez maintenant une bonne idée de ce qu'il faut faire pour sécuriser l'accès au programme. Vous pouvez transformer le programme en une véritable forteresse en multipliant les mesures de sécurité. La seule limite est votre imagination. La plupart des livres qui traitent de la sécurité informatique commencent tous par la même ligne : "n'accordez aucune confiance aux utilisateurs". Il faut bien sûr trouver un juste milieu : la sécurité ne doit jamais être contraignante. Aussi ne tombez pas dans les travers consistant à demander quinze mots de passe à un utilisateur. Il risquerait d'être frustré et d'abandonner le programme.

Chapitre 7

Enregistrer des données

Dans tous les programmes que vous avez réalisés jusque-là, lors de la fermeture du programme, toutes les informations saisies sont effacées, ce qui est gênant. Heureusement, il existe des moyens de sauvegarder des données une fois l'exécution d'un programme terminée. Vous allez voir deux méthodes : la première repose sur des fichiers de sauvegarde ; la deuxième, fondée sur les bases de données, est plus complète mais plus longue à mettre en place.

7.1. Les fichiers

Chacune de ses deux méthodes possède ses avantages et ses inconvénients. Dans le cas de la sauvegarde par fichier, il faut prendre le temps de définir un système de lecture et d'écriture, réfléchir à l'ordre dans lequel vous allez stocker des informations dans le fichier et l'ordre dans lequel vous allez les lire. Il est courant d'utiliser cette méthode lorsque les informations à stocker ne sont pas nombreuses. En effet, un fichier contenant seulement quelques lignes prend beaucoup moins de place qu'une base de données.

7.2. Les bases de données

Lorsque les informations sont plus nombreuses, on veut généralement les organiser. Prenons le cas de recettes de cuisine. Supposons que vous disposiez de cinq recettes : un gâteau, un plat, deux salades et un clafoutis. Il est facile de les garder toutes dans un tiroir, et chaque fois que l'envie de cuisiner vous saisit, de prendre celle qui vous intéresse. Supposons maintenant que vous avez réuni cinq cents recettes. Il devient dès lors nécessaire de les ranger dans une sorte de classeur, par thèmes, par types de plats ou encore par pays d'origine. Une base de données est simplement le classeur dans lequel vous allez les organiser. Il s'agit d'un système de regroupement et d'organisation des données à la manière d'une bibliothèque.

Sur une feuille de papier, dessinez un rectangle plus haut que large. En haut, inscrivez "Recettes". Tirez un trait juste en dessous. Divisez le reste du tableau pour avoir deux colonnes sous la ligne "Recettes". Dans la colonne de gauche, inscrivez tout ce qui peut caractériser une recette de cuisine. Dans celle de droite, inscrivez de quels types de données informatiques ces critères relèvent. Par exemple, le nombre de personnes pour lequel est prévue la recette peut être représenté dans un

programme par un nombre entier. Voici un exemple de tableau que vous pourriez obtenir :

Tableau 7.1 : *Réflexion autour des recettes*

Donnée	Type
Nombre de personnes	Entier
Temps de préparation	Entier
Nom de la recette	Chaîne de caractères
Catégorie de plats	Chaîne de caractères
Recette	Texte

Cette représentation va servir à établir ce que sera une table dans la base de données. Une table est une partie de la base qui va contenir des informations. Ce que vous venez de faire s'appelle la modélisation de la base de données. En effet, une base de données est organisée sous forme de tables. Dans la table des recettes par exemple, chaque recette va représenter une entrée. Une fois en mémoire, la table consistera en un tableau dont chaque ligne sera une entrée. Une fois rempli, le tableau se présentera comme ceci pour le programme :

Tableau 7.2 : *Table des recettes*

Nombre de personnes	Temps de préparation	Nom de la recette	Catégorie de plats	Recette
4	30 minutes	Clafoutis	Dessert	Dans un grand plat…
2	2 minutes	Salade verte	Entrée	Dans un petit plat…

En quoi cette organisation sous forme de tableau fait-elle gagner du temps ? Si vous avez deux recettes, cela ne sert à rien car la quantité d'informations n'est pas importante. Si vous avez deux mille recettes à trier, une base de données va vous permettre de retrouver la recette que vous cherchez parmi les deux mille disponibles, et ce en moins d'une seconde, grâce à un langage spécifique de requêtes. Rassurez-vous, il n'est pas question que vous appreniez le langage des bases de données pour le moment. Visual Basic se charge de traduire pour vous. Pour le moment, retenez ce principe : de manière à optimiser le programme à l'aide d'une base de données, il faut d'abord réfléchir à l'organisation de

la base, puis remplir la base à l'aide de données sous forme de tableau et enfin créer une requête de manière à retrouver rapidement les informations de la base. Si vous vous sentez légèrement perdu, rien de grave, nous éclaircirons tout cela par la pratique dans un instant.

Maintenant que vous connaissez les deux méthodes possibles pour sauvegarder des données, passez à la pratique. Vous allez créer cet utilitaire de gestion de recettes.



Rapidité

Les bases de données étant des fichiers organisés, il est souvent plus rapide de rechercher une information dans de telles structures de données que dans un fichier sur un disque.

7.3. Écrire dans un fichier

La première méthode que vous allez mettre en œuvre est la sauvegarde par fichier. Son point commun avec la méthode des bases de données est de demander un instant de réflexion quant à son usage. Dans ce cas, vous allez sauvegarder chaque recette dans un fichier différent.

1 Commencez par créer un projet dans Visual Basic.

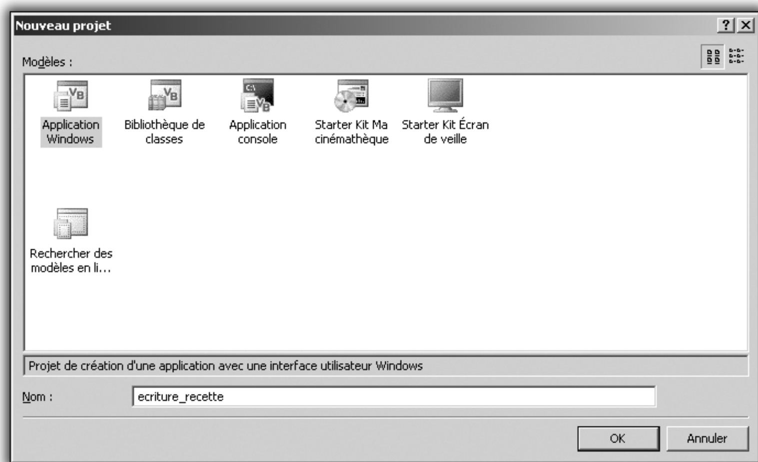


Figure 7.1 : *Projet d'écriture de recettes*

2 Dans le designer de formulaires, ajoutez au formulaire par défaut un contrôle *MenuStrip* puis placez dessus les éléments standard.



Pour savoir comment procéder, lisez le chapitre ***Penser à l'utilisateur****.*



Figure 7.2 : *Formulaire et menu*

3 Dans le formulaire, ajoutez trois *Label* et quatre *TextBox*. Dans la fenêtre des propriétés, changez les propriétés *Name* des *TextBox* en `nombrepers`, `tempsprep`, `titre` et `recette`.

La première va contenir le nombre de personnes pour lequel est prévue la recette, la deuxième le temps de préparation, et les deux autres respectivement le titre et le corps de la recette.

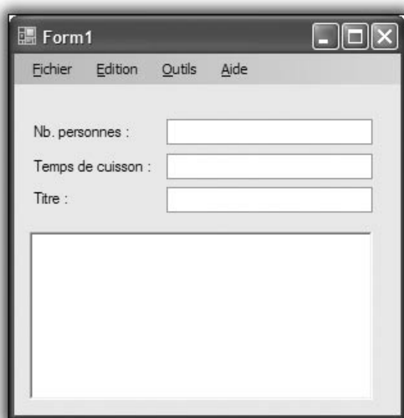


Figure 7.3 : *Prêt à cuisiner*

Tout n'est pas fini. Pour sauvegarder les données, vous allez utiliser des fichiers contenant du texte sous cette forme :

```
Nombre de personnes
Temps de préparation
Titre
Corps de la recette sur plusieurs lignes
```

De cette manière, vous stockerez une information importante par ligne, et il sera aisé de retrouver automatiquement toutes les informations nécessaires quand vous aurez besoin d'ouvrir le fichier contenant la recette.

Il s'agit maintenant que le programme sauvegarde la recette dans un fichier.

4 Ajoutez au formulaire un contrôle de type *SaveFileDialog*.

Dans le designer de formulaires, cliquez sur le menu **Fichier** puis double-cliquez sur le sous-menu **Enregistrer**.

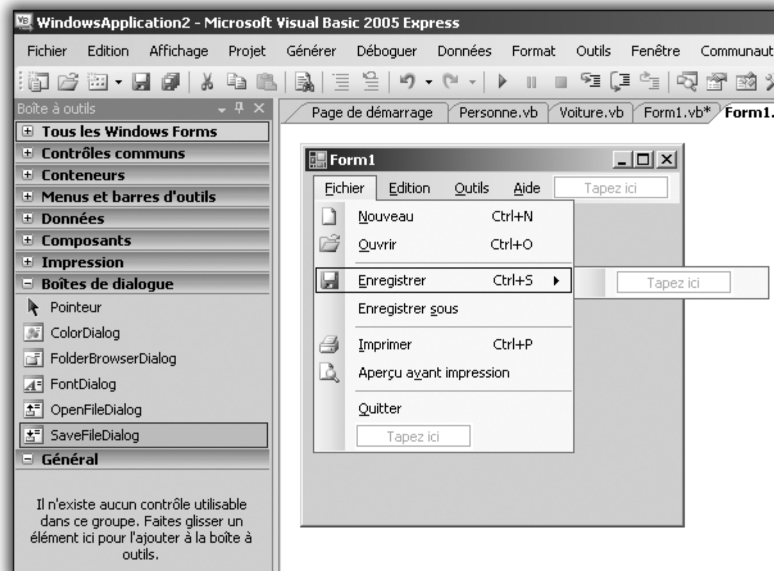


Figure 7.4 : *Méthode d'enregistrement*

Une fois rendu dans l'éditeur de code, recopiez le code suivant :

```
SaveFileDialog1.Filter = "Fichiers TXT(*.txt)|.txt"
If SaveFileDialog1.ShowDialog =_
 Windows.Forms.DialogResult.OK Then
 Dim sw As _
New System.IO.StreamWriter(SaveFileDialog1.FileName)
  sw.WriteLine(NbPers.Text)
  sw.WriteLine(TempsPrep.Text)
  sw.WriteLine(Titre.Text)
  sw.Write(Recette.Text)
  sw.Flush()
 End If
```

Vous précisez en première ligne le type de fichier à sauvegarder. En précisant TXT, vous désignez un fichier texte qui sera réutilisable dans n'importe quelle application.

Le test conditionnel permet de vérifier que l'utilisateur a bien cliqué sur le bouton OK de la boîte de dialogue de sauvegarde de fichier. Une fois que vous êtes sûr qu'un fichier a bien été sélectionné, vous exploitez une nouvelle notion, que sont les flux. Un peu à la manière d'un tuyaux, un flux permet de faire passer de l'information d'un point à un autre, comme une rivière permet à un bateau de naviguer d'une ville à un autre. Vous utilisez ici un objet de type StreamWriter, que l'on peut traduire par "flux d'écriture". Vous faites passer de l'information d'un point A à un point B en l'écrivant une fois arrivé au point B. Dans ce cas, le point A représente chacune des *TextBox* que vous allez remplir, et le point B représente le fichier d'arrivée dans lequel vous voulez enregistrer la recette.



Création du fichier

Lors de la création du fichier, il faut vérifier la place qu'il va prendre sur le disque dur. Si l'espace disque n'est pas suffisant, vous devrez gérer l'erreur.

Vous initialisez le point B (votre fichier) en récupérant dans la boîte de dialogue de sauvegarde le résultat de la sélection de l'utilisateur, en l'occurrence le fichier qu'il a choisi. Ensuite, vous utilisez la méthode WriteLine du flux d'écriture pour ajouter, ligne par ligne, le contenu des différentes *TextBox* du formulaire.

La dernière ligne est un peu particulière. Il s'agit en effet de vider complètement le flux d'écriture, de manière à ne perdre aucune

information à l'intérieur. Cette action a pour but de tout faire arriver au point B, un peu comme si l'on essayait de transférer de l'eau d'une bouteille dans un seau. Le flush permet de secouer la bouteille pour ne laisser aucune goutte à l'intérieur. La différence est que le flux d'écriture fonctionne comme une bouteille dans laquelle l'eau peut rester accrochée. Il faut donc secouer pour la faire tomber. Si vous enlevez la ligne `sw.Flush()`, aucune partie du texte ne sera écrite dans le fichier.



Flush

Le « flush » est l'action de vider complètement un flux d'écriture. Cette opération permet réellement de terminer l'écriture de données.

Comme vous pouvez le voir, l'écriture dans un fichier est très simple et se fait grâce à un flux d'écriture.



Pour rendre cette application plus agréable à utiliser, reprenez les consignes du chapitre ***Penser à l'utilisateur*** *en matière de design d'interface.*

Il faut supprimer les menus inutiles et faire en sorte que le redimensionnement du formulaire principal ne déplace pas les éléments qu'il contient.

Amusez-vous à entrer quelques recettes de manière à disposer d'un répertoire contenant une dizaine de fichiers.

Passons maintenant à la lecture des fichiers de recette et à leur chargement dans l'application.

7.4. Lire un fichier

Votre application permet de sauvegarder des recettes dans un répertoire. Pour que le projet soit complet, il faut maintenant un moyen de récupérer ces informations. Vous allez donc créer un deuxième projet pour lire les recettes dans le répertoire de sauvegarde.

1 Créez un nouveau projet de type application Windows et appelez-le `Lecture Recette`.

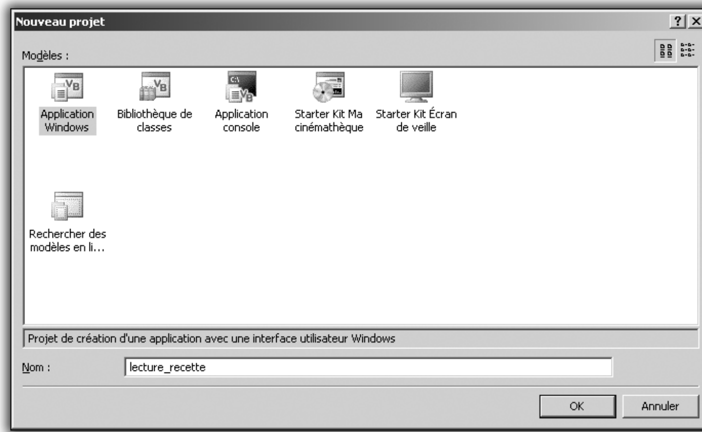


Figure 7.5 : *Projet de lecture des recettes*

2 Dans le designer de formulaires, ajoutez au formulaire principal un contrôle *MenuStrip* ainsi que trois contrôles *Label* et une *TextBox*. Dans la fenêtre des propriétés de la *TextBox*, changez la propriété *ReadOnly* en la mettant à la valeur *True*, afin de rendre la *TextBox* non éditable. En effet, vous voulez uniquement lire les informations contenues dans un fichier.

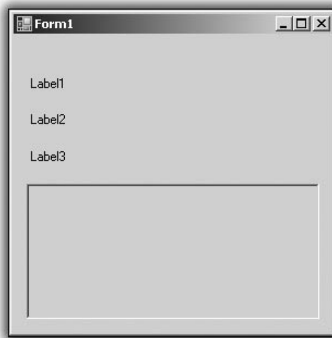


Figure 7.6 : *Squelette de l'application de lecture*

3 Maintenant que vous avez le prototype, modifiez le nom des contrôles pour arriver à ce que vous voulez. Les contrôles *Label* vont contenir chacun une des informations de la recette, à savoir le titre, le nombre de personnes pour lequel la recette est prévue et le temps de préparation.

4 Ajoutez maintenant un contrôle de type *OpenFileDialog* qui va permettre de sélectionner les recettes.

Il faut passer aux instructions. Comme pour l'écriture dans un fichier, la lecture d'un fichier passe par l'utilisation de flux. La différence cette fois est que le flux va dans l'autre sens, à savoir d'une ressource (votre fichier) vers votre programme. Le fonctionnement est le même, seul le nom va changer. Vous allez en effet utiliser un objet de type `StreamReader`, et non un `StreamWriter`. Dans cet exemple, nous avons renommé les *Label* en fonction des informations qu'ils allaient recevoir.

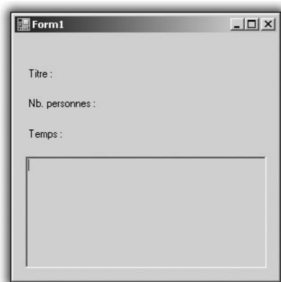


Figure 7.7 : *L'interface finale*

5 Dans le contrôle *MenuStrip*, cliquez sur le menu **Fichier** pour faire apparaître les sous-menus et double-cliquez sur **Ouvrir**.

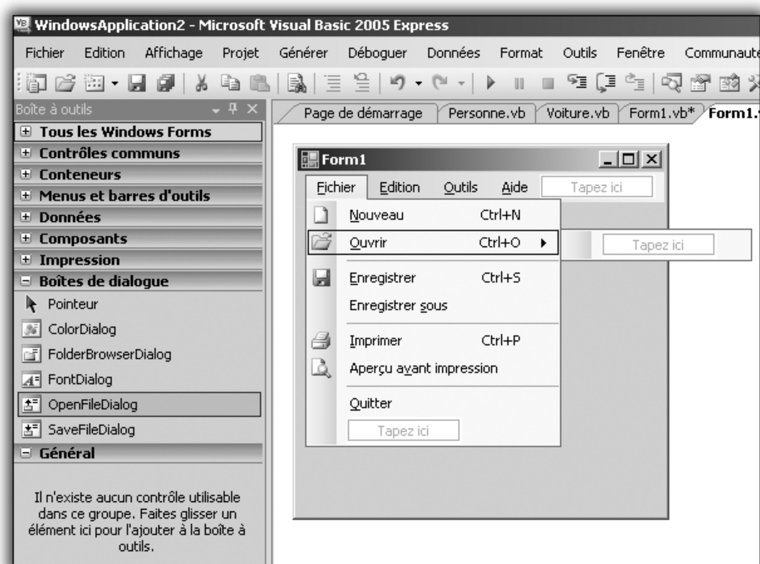


Figure 7.8 : *Méthode d'ouverture*

Une fois dans l'éditeur, ajoutez le code suivant :

```
If OpenFileDialog1.ShowDialog =_
 Windows.Forms.DialogResult.OK Then
 Dim sr As _
New System.IO.StreamReader(OpenFileDialog1.FileName)
 Titre.Text = sr.ReadLine()
 NbPers.Text = sr.ReadLine
 TempsPrep.Text = sr.ReadLine
 CorpsText.Text = sr.ReadToEnd
 sr.Close()
End If
```

Comme dans l'exemple précédent, vous utilisez le premier test de comparaison pour vérifier que l'utilisateur a bien cliqué sur le bouton OK de la boîte de dialogue. Ensuite, vous déclarez un nouveau flux de lecture, qui sera initialisé avec le contenu du fichier sélectionné dans la boîte de dialogue d'ouverture de fichier.

Vous lisez ensuite les trois premières lignes, qui vont respectivement contenir le titre, le nombre de personnes prévu et le temps de préparation. Une fois que ces trois lignes sont lues, il suffit de lire le fichier jusqu'à la fin pour avoir le corps de la recette.

Pour commencer la collection de recettes, voici celle du clafoutis aux cerises :



Le clafoutis aux cerises

Pour 6/8 personnes, temps de préparation : 10 minutes.

Ingrédients :

700 g de cerises bien mûres

2 œufs + 2 jaunes d'œufs

5 cuillerées à soupe de farine (100 g)

5 cuillerées à soupe de sucre roux (100 g)

1/4 de litre de lait (25 cl)

60 g de beurre

1 pincée de sel

1 cuillerée à café d'extrait de vanille

1 sachet de sucre vanillé (facultatif)



Préparation :

Lavez et équeutez les cerises.

Allumez votre four à thermostat 6/7 (180/200° C).

Dans un plat à tarte, mettez le beurre puis enfournez-le.

Battez les œufs en omelette, ajoutez la pincée de sel ainsi que le sucre.

Mélangez bien (au fouet ou au robot).

Ajoutez l'extrait de vanille, mélangez et jetez la farine en pluie.

Mélangez bien jusqu'à obtenir un liquide épais et onctueux.

Sortez le beurre fondu du four, ajoutez-le au liquide puis ajoutez le lait tout en remuant.

Une fois que vous avez mélangé, ajoutez les cerises à la préparation.

Avec un essuie-tout, étalez le beurre resté dans le plat pour le graisser, puis ajoutez la préparation.

Ajoutez quelques noisettes de beurre par-dessus.

Enfournez 45 mn (jusqu'à ce que le dessus commence à bien dorer).

Dès la sortie du four, saupoudrez de sucre vanillé.

Servez tiède.

Utilisez maintenant le programme d'entrée de recettes pour sauvegarder cette recette de clafoutis dans un fichier, puis lancez le programme de lecture de recette pour récupérer le contenu du fichier.

En cliquant sur **Ouvrir** puis en sélectionnant le fichier de la recette de clafoutis, vous obtenez le résultat suivant :

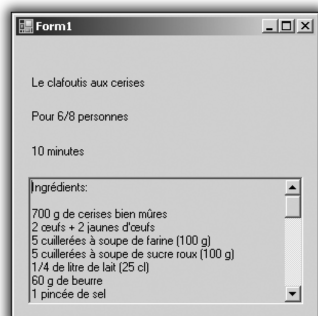


Figure 7.9 :
Chargement de la recette

Vous remarquez un petit problème ? La recette n'est pas du tout mise en page et le texte est difficilement lisible du fait qu'il n'y a aucun retour à la ligne. Pour remédier à cela, il suffit de remplacer le contrôle *TextBox* par une *RichTextBox*, offrant beaucoup plus de fonctionnalités.



Pour en savoir plus à ce sujet, lisez le chapitre ***Penser à l'utilisateur****.*

7.5. Ajouter une base de données au projet

Maintenant que les fichiers n'ont plus de secrets pour vous, vous allez voir la deuxième méthode, consistant à sauvegarder des données dans une base. Les bases constituent un ensemble de fichiers organisés et regroupés en un seul bloc. Pour utiliser une base, il faut un moteur d'exploitation ou gestionnaire. Ici, vous allez utiliser SQL Server Express, qui, tout comme Visual Basic Express, est gratuit et disponible en téléchargement sur le site de Microsoft. Vous le trouverez à l'adresse http://msdn.microsoft.com/vstudio/express/sql/. Téléchargez le programme d'installation et lancez-le. Suivez les instructions comme lors de l'installation de Visual Basic Express. L'opération ne prend que quelques instants.

Une fois SQL Server Express installé, vous n'avez plus qu'à l'utiliser à travers Visual Basic Express. Vous allez maintenant reprendre le programme de gestion des recettes, en le rendant beaucoup plus performant par l'utilisation d'une base de données SQL Server Express.

1 Commencez par créer un nouveau projet de type application Windows dans Visual Basic Express.
2 Une fois dans le designer de formulaires, cliquez du bouton droit sur le nom du projet dans l'Explorateur de solutions puis cliquez sur **Ajouter un nouvel objet**.
3 Choisissez *Base de données SQL* (SQL Database).

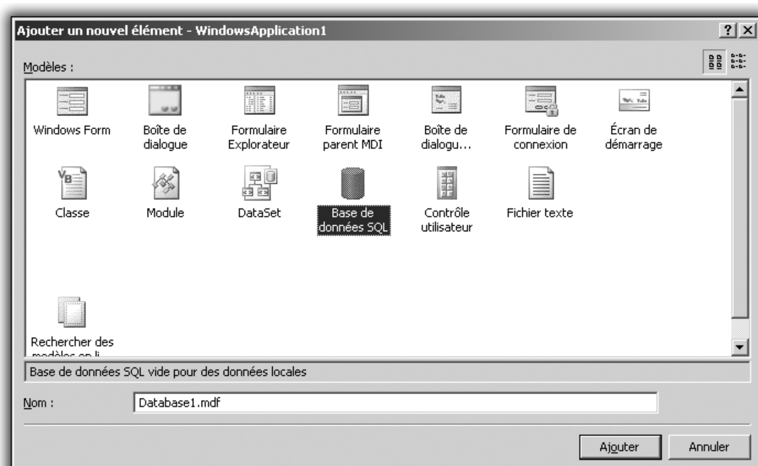


Figure 7.10 : *Sélection d'une base*

4 Une fois la base sélectionnée, donnez-lui le nom `recettes.mdf` puis cliquez sur le bouton **Ajouter**. L'Assistant de configuration de source de données s'affiche.

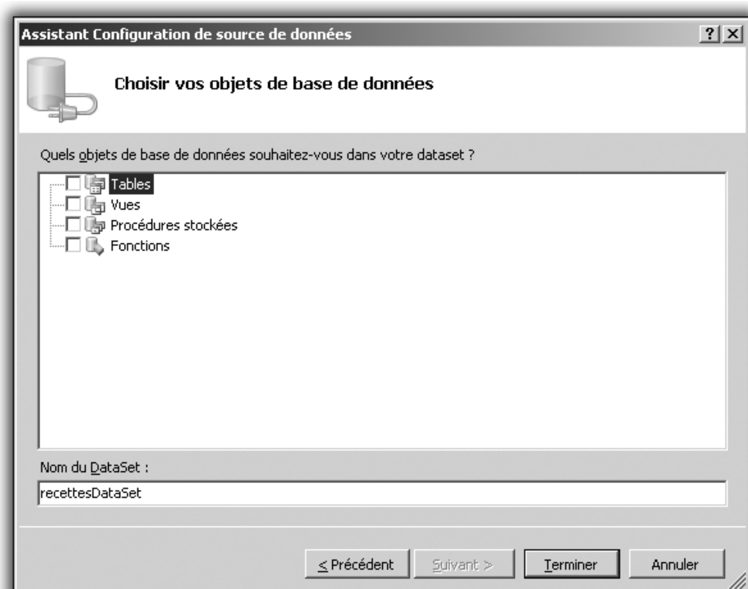


Figure 7.11 : *Assistant de configuration de source de données*

5 Ignorez cet Assistant et cliquez sur **Annuler**. L'Explorateur ressemble maintenant à la capture suivante, avec le formulaire principal et la base de données.

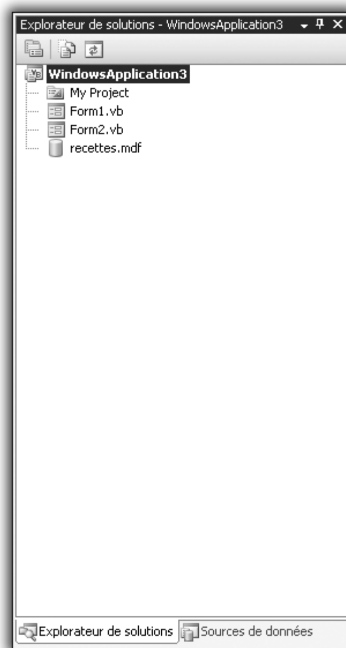


Figure 7.12 :
La base est ajoutée

Une base de données est une sorte de classeur dans lequel vous pouvez à loisir incorporer des feuilles que l'on appelle "tables" et qui permettent d'organiser les données. Il faut maintenant réfléchir à la structure de ces tables et à la manière dont vous allez organiser les données. Le but ici est de ne pas répéter les données plusieurs fois dans le cas où vous auriez plusieurs tables. Le programme étant plutôt simple, vous avez besoin d'une seule table que vous allez représenter sous forme de tableau, comme décrit en introduction.

Un point important est que cette base de données est complètement indépendante de Visual Basic Express ou du langage Visual Basic, comme vous allez le voir maintenant.

Il faut à présent ajouter une table à la base. À gauche de l'interface de Visual Basic, un nouvel onglet est apparu, nommé **Explorateur de bases**.

6 Cliquez sur l'onglet **Explorateur de bases** pour développer la fenêtre afférente.



Figure 7.13 : *L'Explorateur de bases*

7 Cliquez du bouton droit sur le dossier *Tables* puis sélectionnez **Ajouter une nouvelle table**.

À la place du designer de formulaires s'affiche un tableau à trois colonnes, la première indiquant le nom de la colonne dans la table, la seconde le type de données que va contenir cette colonne et la troisième une boîte à cocher pour spécifier si oui ou non vous voulez que cette valeur puisse être nulle.

8 Vous allez vous conformer au schéma de table spécifié en introduction : une première colonne de la table représentera le nombre de personnes pour lequel la recette est prévue, une deuxième colonne le temps nécessaire, une troisième le nom de la recette, une quatrième la catégorie de plats et une cinquième le corps de la recette. Vous allez donc commencer par entrer les

spécifications de la première colonne, à savoir le nombre de personnes. Dans la case *Column Name*, entrez la valeur `NbPers`. Dans la colonne *DataType*, vous n'avez rien à entrer ; vous devez faire une sélection parmi une liste de types possibles. Pour symboliser un nombre entier simple, utilisez *numeric(18,0)*. Le tableau s'agrandit en ajoutant automatiquement une seconde colonne à remplir. Remplissez les lignes suivantes comme vous venez de le faire, en ajoutant une ligne `TempPrep` de type *ntext*, une ligne `Titre` de type *ntext*, une ligne `CatPlat` de type *Text*, et une ligne `CorpsRec` de type *text*. La différence entre les types *text* et *ntext* réside dans la longueur admissible des données que l'on peut entrer dans chaque case.

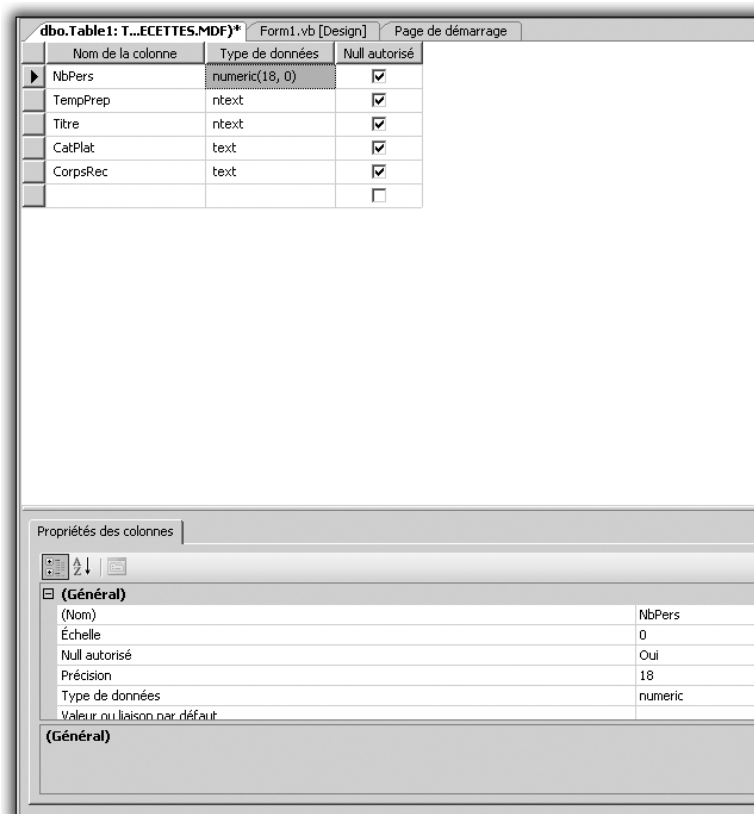


Figure 7.14 : *Définition de la table*

9 Sauvegardez maintenant la table que vous venez de créer en appuyant sur les touches Ctrl+S.

10 Un nom vous est alors demandé pour la sauvegarde de la table. Entrez `MaTable` puis cliquez sur le bouton OK.

La table s'ajoute alors dans l'Explorateur de bases.

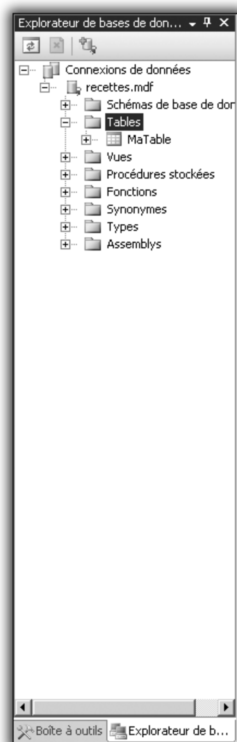


Figure 7.15 :
L'ajout de la table

Vous allez maintenant ajouter des valeurs dans la table à partir de Visual Basic. Dans l'Explorateur de solutions, cliquez du bouton droit sur la table que vous venez d'ajouter puis sélectionnez **Afficher les données de la table**. Le designer montre alors un tableau qui représente la table, les colonnes étant celles que vous venez d'entrer (voir Figure 7.16).

Ajoutez sur la première ligne les informations de la recette du clafoutis. Pour le nombre de personnes, faites attention à entrer un nombre entier pour respecter le format de type *numeric*. Vous êtes en effet moins libre que lors de l'utilisation des fichiers. Mais comme vous le verrez plus tard, cela permet de faciliter les recherches. Libre à vous d'ajouter d'autres lignes pour représenter d'autres recettes. Une fois que vous avez ajouté les recettes que vous voulez, passez à la section suivante.

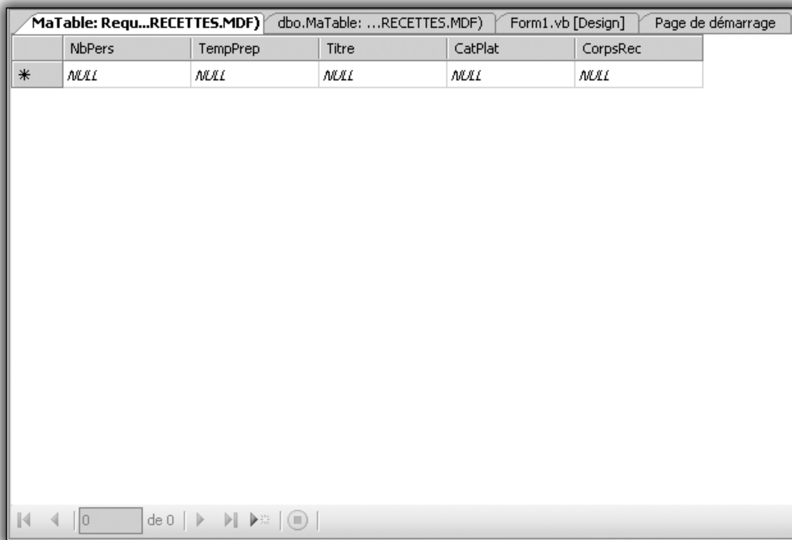


Figure 7.16 : *Le designer de tables*

7.6. Afficher les données de la base

Vous n'aurez jamais à saisir toutes les données à travers Visual Studio. Vous disposez d'outils pour afficher, modifier et rechercher les éléments dans la base. Visual Basic .NET permet en effet la manipulation des bases de données grâce à ADO .NET, qui peut être vu comme un ensemble de fonctions, méthodes et objets qui permettent d'effectuer toutes les opérations possibles sur les bases de données. Vous allez maintenant afficher les informations de la base de données dans le formulaire.

Revenez au designer de formulaires. Juste à côté de l'Explorateur de solutions, notez la présence de l'onglet **Sources de données**.

1 Cliquez sur l'onglet **Sources de données**, qui s'affiche à la place de l'Explorateur de solutions.

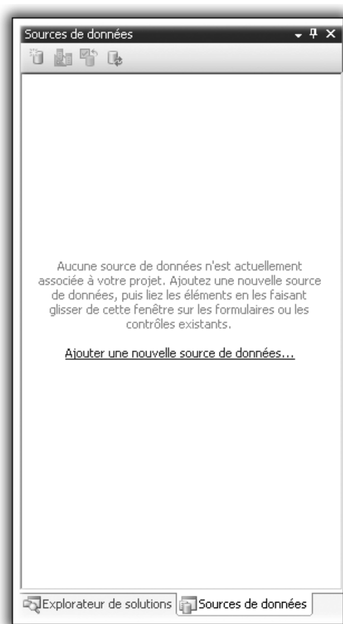


Figure 7.17 : *Les sources de données*

2 Cliquez sur le lien qui vous propose d'ajouter une source de données au projet. Choisissez *Base de données* et cliquez sur **Suivant**.

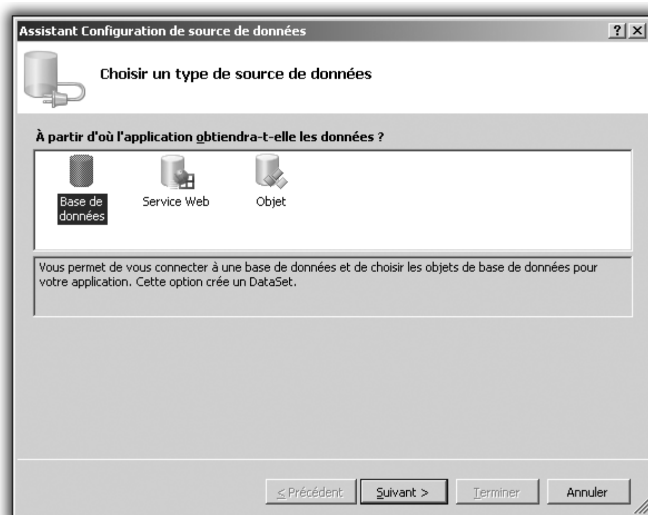


Figure 7.18 : *Le choix de la source*

L'écran suivant permet de choisir la base de données à utiliser comme source. Normalement, celle créée plus tôt et nommée *Recettes.mdf* est déjà sélectionnée.

3 Cliquez sur **Suivant**.

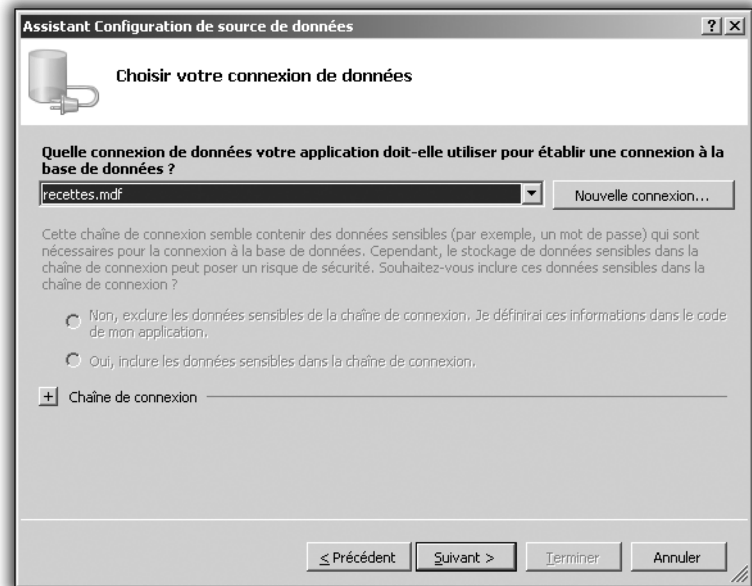


Figure 7.19 : *Sélection de la base*

4 Il vous est ensuite demandé si vous voulez créer ce qu'on appelle une chaîne de connexion. Répondez oui à cette question. Créer une chaîne de connexion permet de stocker quelque part dans le programme un lien qui pointe vers la source de données. Ainsi, si jamais la source de données change d'endroit, il suffira d'éditer le lien dans la configuration de l'application au lieu de reprendre toute la configuration de la base de données. Vous gagnerez ainsi un temps précieux. Répondez donc par l'affirmative puis cliquez sur **Suivant**.

5 Il vous est ensuite demandé quelles données de la source vous voulez reprendre. Vous voulez afficher vos recettes. Il faut donc reprendre toutes les informations de la table. Cliquez donc sur les boîtes de choix, comme indiqué sur l'écran suivant :

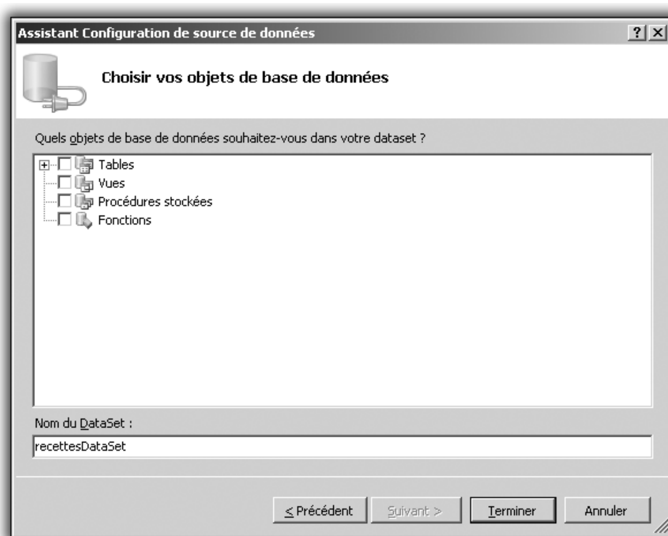


Figure 7.20 : *Récupération de la table*

6 Cliquez sur le bouton **Terminer**. La configuration de la source de données est complète.

La configuration complète de la source de données sert simplement à établir un lien entre la source de données, la base et le programme. Maintenant que vous avez un *DataSet* fonctionnel, il faut l'ajouter au formulaire. Retournez dans le designer de formulaires.



Le DataSet

Le *DataSet* est un jeu de données qui fera pour vous le lien entre la base et les données recherchées. Vous pouvez l'éditer à tout moment dans la fenêtre des sources de données.

À partir de maintenant, dans la fenêtre qui montre les sources de données, lorsque vous cliquez sur le nom de la table, vous verrez apparaître sur la droite une icône de liste déroulante qui propose deux choix : *DataGridView* ou *Details*. Ces deux choix concernent la représentation des données de la base sur le formulaire. Pour un premier exemple, choisissez *DataGridView* et faites glisser le nom de la table vers le formulaire. Un tableau est ajouté au formulaire, ainsi que des contrôles de navigation et un contrôle *MaTableBindingSource*, qui est le

lien physique entre le formulaire et la table. Voyons maintenant comment utiliser tout cela.

Lancez une première fois l'application.

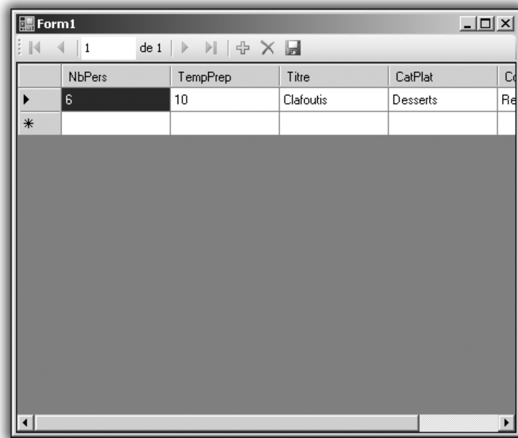


Figure 7.21 : *L'application*

La ligne relative au clafoutis s'affiche bien en première ligne. Observez les boutons situés dans la barre d'outils en haut de la fenêtre d'application. Ils servent à naviguer parmi les enregistrements de la base de données. Cliquez sur le signe **+** situé parmi les outils. Une ligne est alors ajoutée au tableau, vous permettant d'insérer un enregistrement dans la base. Ajoutez une salade par exemple. Cliquez sur l'icône en forme de disquette à droite de la barre d'outils pour enregistrer les changements. Quittez l'application. De retour dans Visual Basic, appuyez une nouvelle fois sur la touche F5 pour relancer le programme. Vous serez déçu en constatant que la modification n'a pas été prise en compte et que la base de données ne contient que la ligne relative au clafoutis.

Ne pas perdre la base

Lorsque que Visual Basic lance le projet, il génère la base de données et celle que vous aviez lors de votre précédente utilisation est écrasée par celle nouvellement générée, qui ne contient bien évidemment pas la modification. Rassurez-vous, il existe un moyen simple de remédier à cela.

1 Revenez dans Visual Basic, dans le designer de formulaires.
2 Cliquez sur la base de données dans l'Explorateur de solutions, de manière à afficher ses propriétés.
3 Dans la liste des propriétés, cherchez celle qui se nomme *Copie vers le répertoire de destination*. Par défaut, sa valeur est *Toujours*.
4 Changez cette valeur par *Ne pas copier*.

Cela fait, si vous lancez le programme, une erreur survient lors du lancement. L'initialisation de la base est impossible. En utilisant cette méthode, il faudra toujours penser à copier la base de données dans le répertoire d'exécution du programme. Normalement, la base est composée de deux fichiers, l'un portant l'extension *.mdf* et l'autre l'extension *.ldf*. Ils sont tous deux stockés à la racine du projet. Par défaut, ce répertoire est *Mes Documents\Visual Studio 2005\Projets\Nom du projet\Nom du projet*.

5 Copiez ces deux fichiers dans le répertoire *Bin\debug* du répertoire de projet.
6 Si vous avez changé la propriété dans l'éditeur de propriétés de la base du designer de formulaires, vous avez fini. Relancez maintenant le programme et tentez une modification en ajoutant une valeur.

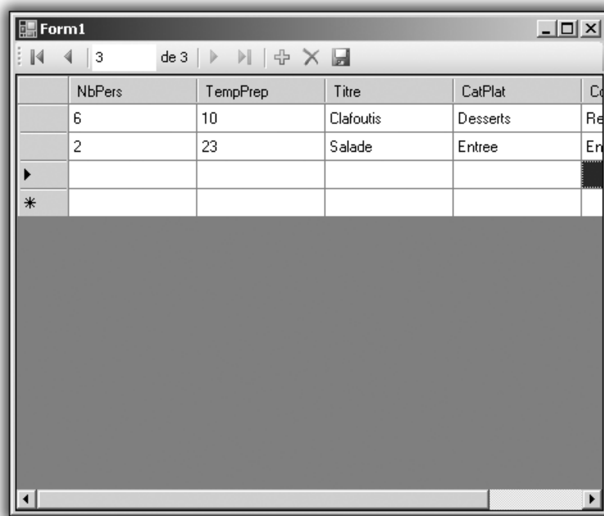


Figure 7.22 : *L'application après l'ajout d'une deuxième ligne*

Fermez maintenant le programme, revenez sous Visual Basic.

7 Affichez les données de la table de recettes et regardez le nombre de lignes.

La ligne concernant la salade n'apparaît pas. Relancez le programme. La ligne apparaît bien dans le tableau de visualisation de la base. Étrange ? Pas du tout. Gardez bien à l'esprit que vous travaillez maintenant sur une copie de la base qui est située dans le répertoire de génération du programme. Les modifications ne sont donc pas synchronisées. Si vous modifiez la base dans Visual Basic, il faudra répercuter les nouveautés en copiant les informations de la base dans le répertoire encore une fois.

8 Retournez maintenant dans le designer de formulaires.

9 Nous avons énoncé plus haut deux manières d'afficher les données d'une base de données. Or vous n'en avez utilisé qu'une. Supprimez du formulaire le *DataGridView*, mais pas les outils de navigation. Dans la fenêtre des sources de données, cliquez sur le nom de la table, puis changez dans la liste déroulante la valeur de représentation en sélectionnant *Details*. La représentation se fera sous la forme d'un lot de contrôles, comme lorsque vous lisez à partir d'un fichier. En cliquant sur chacun des champs de la table, vous pouvez voir qu'ils disposent de la même petite icône de choix qui permet de sélectionner le type de contrôle qui les représentera. Choisissez *TextBox* pour chacun des contrôles pour le moment.

10 Dans le designer de formulaires, arrangez les contrôles comme bon vous semble, puis générez le projet en appuyant sur F5.

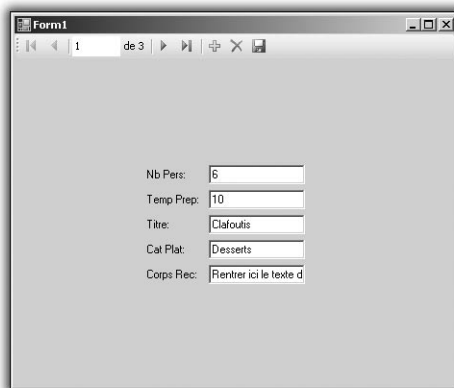


Figure 7.23 : *La vue Details*

Chaque champ est rempli avec la valeur associée dans la base. Utilisez les flèches droite et gauche situées dans la barre de menus pour naviguer dans les fiches de recette. Comme vous l'avez fait dans le tableau, vous pouvez ajouter une entrée en cliquant sur le bouton **+** de la barre d'outils et enregistrer les changements dans la base.

7.7. Aller plus loin grâce aux bases de données

Vous avez vu dans ce chapitre dédié a la sauvegarde de données qu'il était facile de sauvegarder les données d'un programme dans une base ou dans un jeu de fichiers manuellement définis. Les possibilités d'utilisation des bases de données dépassent le cadre de ce livre, mais sachez que vous pouvez avoir plusieurs tables dans une base et recouper les données entre elles. Dans l'application de gestion des recettes, cela permettrait par exemple de les classer, d'ajouter des fonctionnalités de recherche, en un mot d'aller plus loin dans la classification.

Gardez à l'esprit une ligne conductrice d'organisation. Le travail le plus long et le plus dur lors de la création d'une base de données est sa modélisation. Il en est de même pour l'organisation des données dans un système de fichiers. L'organisation de la base, pour être performante, nécessite un long moment de réflexion.

Efforcez-vous de reprendre cette application en améliorant la base de données et sa représentation dans le formulaire. Essayez également d'imaginer un moyen d'exporter les données de la base dans des fichiers pour assurer au programme une souplesse totale.

Chapitre 8

Rendre un programme robuste

Un jour ou l'autre, vous devrez faire face à des bogues ou à des erreurs de programmation. Il n'existe pas de programmeur qui n'a jamais fait une seule erreur. Forcément, vous en ferez, mais il ne faut pas vous décourager pour autant. Le tout est de les prévoir et de les éliminer pour une utilisation réelle du programme.

Vous allez voir dans ce chapitre les différents éléments qui vous permettront de retrouver et d'éliminer ces erreurs. En ce sens, vous utiliserez des exceptions. Elles permettent de détecter les erreurs lors de la programmation et d'effectuer des opérations en conséquence. Si elles n'ont pas toutes été recensées lors de cette étape, vous apprendrez à les détecter lors de la phase de débogage, et aussi à créer des traces écrites du déroulement du programme, de sorte que vous pourrez le contrôler a posteriori. Nous finirons par quelques habitudes de programmation à prendre, à savoir insérer des commentaires et rédiger la documentation, qui permettront de minimiser ces problèmes.

8.1. La prévention maximale

Le mot d'ordre du programmeur pourrait être "toujours prévoir au maximum". Beaucoup d'erreurs de programmation et de comportements non voulus sont dus au fait que la personne qui a programmé n'a pas prévu tel ou tel cas de figure. Prenons un exemple :

```
Dim age As Integer
age = -1
If (age < 18) Then
  MessageBox.Show("Vous êtes mineur")
Else
  MessageBox.Show("Vous êtes majeur")
EndIf
```

Le programme va dire que l'on est majeur, alors qu'en fait, l'âge donné n'est pas valable. En effet, il n'existe pas d'âge négatif. Une solution est de prévoir ce cas. On peut par exemple considérer qu'un âge est valide s'il est compris entre 0 et 130 (pour ceux et celles qui ont une grande espérance de vie !). Voici le bout de programme corrigé :

```
Dim age As Integer
age = -1
If (age < 0 || age > 130) Then
  MessageBox.Show("L'âge donné n'est pas valide")
ElseIf (age < 18) Then
    MessageBox.Show("Vous êtes mineur")
Else
```

```
    MessageBox.Show("Vous êtes majeur")
  EndIf
EndIf
```

Maintenant le programme vérifie si l'âge donné est valide et donne une réponse juste, alors que dans le cas précédent, il a donné une réponse fausse.

Il s'agit là d'une manière de minimiser les erreurs, simple et relativement efficace. Cependant, Visual Basic .NET intègre des mécanismes plus puissants pour cette tâche : les exceptions.

Gestion des exceptions

Un bon recensement des cas de figure possibles permet de limiter les comportements non voulus du programme. Cependant, lors de l'exécution, il s'avère parfois que des cas problématiques n'ont pas été détectés, même si le programmeur a fait preuve d'une grand vigilance. Visual Basic .NET intègre un système de gestion d'exceptions qui permet de gérer de manière propre ces erreurs d'exécution.

Les exceptions sont des instances de classe, dérivées de la classe `System.Exception`. Pour comprendre leur fonctionnement, vous n'avez pas besoin de connaître les notions sous-jacentes de programmation.



*Ces notions sont décrites au chapitre **Passer au niveau supérieur**.*

Disons pour le moment que ce sont simplement des enregistrements, avec des attributs dont les plus importants sont :

- `Message` : c'est une chaîne de caractères qui donne une description de l'exception qui a été levée.
- `StackTrace` : donne des informations concernant la fonction qui a provoqué l'exception, le dossier, le fichier, et même la ligne correspondante.

Voici le fonctionnement de base d'une exception :

Lorsqu'il y a une erreur d'exécution, c'est-à-dire une erreur qui n'a pas été détectée à la programmation, une exception correspondant à l'erreur

est levée. Le programme s'arrête. En réalité, il ne s'arrête pas vraiment, mais disons que les instructions normales ne sont plus exécutées. Il va se dérouler jusqu'à trouver une structure capable de gérer l'exception, à savoir un bloc `catch` (nous y reviendrons). S'il n'est pas dans la fonction en cours, le programme va continuer dans les fonctions suivantes pour voir si un tel bloc existe. S'il n'existe pas, il y a un gestionnaire par défaut, qui affiche le type de l'exception, d'éventuelles informations supplémentaires, et l'application se termine. Il se présente comme ceci :

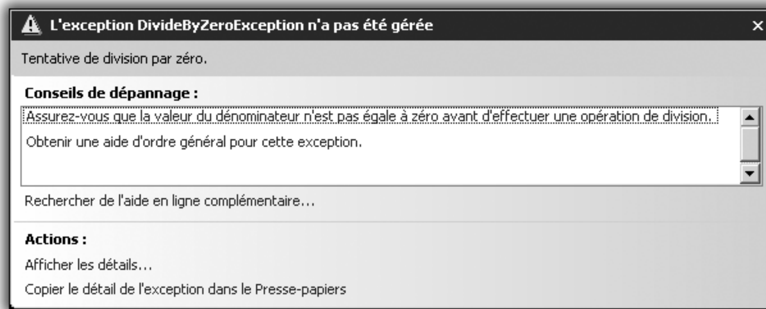


Figure 8.1 : *Une exception non gérée*

Généralement, il ne faut pas utiliser le gestionnaire par défaut. C'est pourquoi vous allez apprendre à créer et à utiliser des structures de gestion d'exceptions.

Structure de gestion d'exceptions : le bloc Try… Catch… Finally

Les exceptions sont des éléments importants pour une bonne programmation. En les utilisant à bon escient, et de manière efficace, vous pourrez créer des programmes robustes, prêts à réagir au moindre souci.

Il est important de faire des structures de gestion d'exceptions dans des portions de programme délicates, comme des divisions par un nombre variable (pour éviter la division par zéro), des accès à des fichiers ou dossiers (pour vérifier leur existence), etc.

Voici la syntaxe d'un bloc `Try… Catch… Finally` :

```
Try
  Instructions_a_evaluer
Catch exception As type_de_lexception
  Instructions_de_gestion_des_exceptions
Finally
  Instructions_toujours_executees
End Try
```

Dans le bloc `Try`, il faut mettre les instructions à évaluer pour vérifier le déroulement du programme. Le bloc sera exécuté et, si aucune exception n'a été levée, le programme continuera normalement.

C'est dans le bloc `Catch` que les exceptions sont traitées. Ici vous pouvez afficher les messages, éventuellement les écrire dans des fichiers texte (vous verrez plus loin les journaux d'événements), modifier les données en conséquence, etc. Dans cette partie, il est intéressant d'utiliser les propriétés `Message` et `StackTrace` décrites précédemment. Il peut y avoir plusieurs blocs `Catch`. Dans ce cas, on parle de traitement sur plusieurs niveaux. On va du plus spécialisé au plus général :

```
Try
  Instructions_a_evaluer
Catch exception1 As type_de_lexception1
  Instructions_de_gestion_des_exceptions1
Catch exception2 As type_de_lexception2
  Instructions_de_gestion_des_exceptions2
Catch exception3 As type_de_lexception3
  Instructions_de_gestion_des_exceptions3
Finally
  Instructions_toujours_executees
End Try
```

Dans le bloc `Finally`, insérez des instructions à exécuter dans tous les cas, qu'une exception ait été levée ou non. Ce bloc est facultatif. La plupart du temps, des instructions à exécuter dans tous les cas l'ont déjà été par ailleurs dans le programme.

Exemple : la gestion de la division par zéro

Rien de tel que la pratique pour comprendre les exceptions. Voici un programme qui montre comment les utiliser.

```
Dim dividende As Integer
Dim diviseur As Integer
dividende = 12
diviseur = 0
Try
```

```
  Dim resultat As Integer
  resultat = dividende / diviseur
  MessageBox.Show("Résultat = " + resultat)
Catch exception As_
 System.DivideByZeroException
  MessageBox.Show(exception.Message)
  MessageBox.Show(exception.StackTrace)
  diviseur = 1
Catch exception As System.Exception
  MessageBox.Show(_
"Il y a eu une exception d'ordre général"_
)
Finally
  Dim resultat As Integer
  desultat = dividende / diviseur
  MessageBox.Show("Résultat final = " + resultat)
End Try
```

Le but de ce programme est d'établir une division par zéro, ce qui permettra de voir comment fonctionnent les exceptions et comment en tirer le meilleur parti.

Tout d'abord, on est obligé d'utiliser des variables pour provoquer cette erreur. On n'aurait pas pu écrire directement `resultat = 12 / 0` car le compilateur aurait directement provoqué une erreur, visible dans Visual Studio :



Figure 8.2 : *Division par zéro constante*

De plus, `dividende` et `diviseur` étant utilisés dans plusieurs blocs, on ne peut pas se contenter de les mettre dans le bloc `Try`, à cause de la portée des variables.

Dans le bloc `Try`, la division de `dividende` par `diviseur`, valant respectivement 12 et 0, va lever une exception de division par zéro, qui va court-circuiter l'instruction censée afficher le résultat de l'opération.

Un bloc `Catch` suit. Le programme peut donc traiter l'exception. Comme il s'agit effectivement d'une division par zéro, elle sera traitée au premier niveau. On affiche le message de l'exception :



Figure 8.3 : *Message de l'exception*

On affiche toutes les informations afférentes via la propriété `StackTrace` :

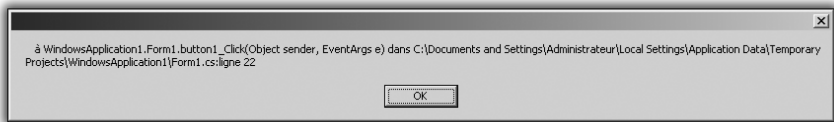


Figure 8.4 : *StackTrace de l'exception*

On modifie `diviseur` étant donné que l'on va l'utiliser ensuite dans le programme. Il ne faudra pas que la même erreur se produise, c'est pourquoi il faut changer sa valeur. Cette exception ayant été traitée au premier niveau, il n'est pas nécessaire de passer au dernier niveau de `Catch`. Le programme va donc au bloc `Finally`.

Dans ce dernier bloc, la même division est refaite, alors que `diviseur` a été modifié. L'opération se passe sans encombre et le programme est en mesure d'afficher le résultat de cette division. À ce moment, une exception a été levée, elle a été traitée, et le programme peut donc continuer normalement, à partir du `End Try`.

À travers cet exemple, vous avez pu voir une utilisation à la fois basique et efficace des exceptions. Apprenez à bien écrire vos programmes en utilisant ce système, car il se révèle efficace pour la gestion des erreurs, surtout quand il est associé à d'autres éléments, comme les journaux d'événements que vous allez voir ci-après. La gestion des exceptions peut aller beaucoup plus loin, par exemple avec le redéclenchement d'exceptions en cascade, mais cela n'entre pas dans le cadre de cet ouvrage. Maîtriser l'aspect simple des exceptions permet d'améliorer la qualité des programmes, et avec la pratique, vous éviterez de nombreux plantages.

8.2. La chasse aux bogues

Dans la section précédente, vous avez vu à travers les exceptions comment vous prémunir des erreurs d'exécution pendant la phase de programmation. Cependant, certaines erreurs se produisent à cause de facteurs indépendants de la programmation. Prenons par exemple le bout de programme précédent :

```
Try
  Dim resultat As Integer
  resultat = dividende / diviseur
  MessageBox.Show("Résultat = " + resultat)
Catch exception As System.DivideByZeroException
  MessageBox.Show(exception.Message)
  MessageBox.Show(exception.StackTrace)
  diviseur = 1
Catch exception As System.Exception
  MessageBox.Show( _
"Il y a eu une exception d'ordre général" _
)
Finally
  Dim resultat As Integer
  desultat = dividende / diviseur
  MessageBox.Show("Résultat final = " + resultat)
End Try
```

Dans ce morceau de programme, on considère que `dividende` et `diviseur` ne sont pas des variables déclarées explicitement, mais issues d'un fichier texte de cette forme :



Figure 8.5 :
Fichier avec des arguments justes

Ce fichier texte donne les valeurs de `dividende` et `diviseur`. Le programme peut être exécuté et avoir le même comportement que si l'on

avait défini directement les variables, dans la mesure où les valeurs entrées sont correctes.

Pour savoir comment enregistrer des données dans un fichier texte, reportez-vous au chapitre *Enregistrer des données*.

Imaginez qu'un petit malin remplace des valeurs correctes de `dividende` et `diviseur` par des chaînes de caractères non numériques.



Figure 8.6 :
Fichier avec de faux arguments

L'opération `resultat = dividende / diviseur` ne peut se faire.

L'erreur provoquée n'étant pas une division par zéro, elle est traitée en tant qu'exception générale et le programme affiche tout simplement un message à ce sujet. Bien que l'erreur soit traitée, les informations ne sont pas particulièrement pertinentes. De plus, une fois ce traitement exécuté, on n'a plus de trace du déroulement du programme.

Pour pallier cette insuffisance, on utilise des journaux d'événements. Ils permettent de conserver des traces de l'exécution du programme (de son comportement). Il est possible ainsi de les examiner a posteriori et de corriger le programme en conséquence, pour le rendre de plus en plus robuste.

Création de journaux d'événements

Les journaux d'événements, ou logs, sont des fichiers texte, écrits dans un format particulier. Ils gardent des traces de l'exécution d'un

programme. Les programmeurs, pour leurs codes les plus sensibles, appliquent une politique de journaux d'événements, qu'ils pourront utiliser pour régler des problèmes ou des litiges.

Par exemple, toutes vos opérations bancaires sont stockées de cette manière. Les logiciels utilisés par les banques ont en place un système de logs qui leur permet de suivre vos opérations. Ainsi, quand vous appelez pour demander des précisions, ils peuvent savoir ce qu'il s'est passé.

Les logs sont importants à partir du moment où les traitements sont sensibles. Même s'ils ne le sont pas, il est quand même bon d'avoir un tel système, car il est gage de fiabilité et offre des possibilités de suivi de manière simple et efficace.

La création de ces journaux passe par l'utilisation des classes `Debug` et `Trace` de la bibliothèque de classes .NET. Il n'est pas nécessaire de connaître les subtilités de la programmation objet pour pouvoir les utiliser.



*La programmation objet est traitée au chapitre **Passer au niveau supérieur**.*

Ces classes existent dans l'espace de noms `Diagnostics`. C'est pourquoi, pour pouvoir les utiliser sans avoir à donner le chemin complet à chaque fois, il faut ajouter `Diagnostics` à l'ensemble des espaces de noms du projet. Ce faisant, vous aurez également accès aux autres classes utilisées pour la création des journaux. Insérez donc cette ligne au début du projet :

```
Using System.Diagnostics
```

Les méthodes d'écritures des journaux

`Debug` et `Trace` sont des classes qui possèdent un certain nombre de fonctions permettant d'écrire différentes informations concernant le programme. Ces fonctions diffèrent par leurs conditions d'écriture et par leur contenu. Elles sont au nombre de six. En voici la description :

- `Write` : permet d'écrire un texte sans condition spéciale.
- `WriteLine` : écrit un texte et termine celui-ci par un retour à la ligne. Cette fonction permet d'écrire effectivement le texte sans

avoir à passer par le mécanisme de `Flush` que nous expliquerons un peu plus loin.

- `WriteIf` : écrit un texte seulement si une certaine condition booléenne est vérifiée.
- `WriteIfLine` : écrit un texte si la condition booléenne passée en paramètre est vraie. Comme `WriteLine`, cette fonction permet d'écrire effectivement le texte sans passer par le `Flush`.
- `Assert` : écrit un message si une certaine condition booléenne est fausse. De plus, le message en question apparaît dans une boîte de dialogue.
- `Fail` : écrit un message et l'affiche dans une boîte de dialogue. Cette fonction est utilisée dans les cas d'échec et d'erreur exclusivement.

Voici maintenant comment les utiliser (on considère que les variables du programme précédent existent) :

- Texte explicatif de l'instruction en cours :

```
Trace.Write("Opération : " + dividende + " / " +
✂ diviseur)
```

- Texte explicatif de l'instruction en cours avec un retour à la ligne :

```
Trace.WriteLine("Opération : " + _ dividende + " / "
✂ + diviseur)
```

- On écrit que l'opération est impossible si le diviseur est égal à zéro :

```
Trace.WriteIf(diviseur == 0, "Opération impossible :
✂ diviseur nul")
```

- On écrit que l'opération est impossible si le diviseur est égal à zéro. On ajoute également un retour à la ligne :

```
Trace.WriteIfLine(diviseur == 0, "Opération
✂ impossible : diviseur nul")
```

- On écrit l'opération en cours, après avoir vérifié qu'elle était valable :

```
Trace.Assert(diviseur != 0, "Opération : " +
✂ dividende + " / " + diviseur)
```

- Il y a eu un échec. On donne l'explication et on l'inscrit. Par exemple, si les opérandes ne sont pas des nombres :

  ```
  Trace.Fail("Les opérateurs de l'opération ne sont pas
  ✂ valides")
  ```

Ces opérations permettent d'écrire les messages dans les journaux, de manière à garder des traces du programme. Il est recommandé d'utiliser les méthodes qui intègrent les retours à la ligne, telles `WriteLine` et `WriteIfLine`, pour ne pas avoir à se préoccuper du problème du `Flush`. La plus importante est `WriteLine`. En effet, vous pouvez gérer les conditions vous-même :

```
If (diviseur == 0)
  Trace.WriteLine("L'opération ne peut être faite _
car le diviseur est nul")
```

Le fait d'afficher une boîte de dialogue avec `Assert` ou `Fail` peut être intéressant pour la compréhension. Mais de la même manière, vous pouvez gérer la condition vous-même. Ce qui suit est équivalent à l'écriture précédente. Cependant, une boîte de dialogue vous préviendra de ce qu'il s'est passé.

```
If (diviseur == 0)
  Trace.Fail("L'opération ne peut être faite car le
  ✂ diviseur est nul")
```

Vous connaissez à présent les méthodes qui permettent d'écrire les journaux d'événements. Il faut maintenant voir où elles vont les écrire. C'est là qu'apparaît la notion d'écouteur, c'est-à-dire la cible d'écriture des logs.

Les écouteurs

Les fonctions vues précédemment permettent d'écrire les informations que vous voulez concernant le déroulement du programme. La question est maintenant de savoir où écrire ces traces. C'est en fait dans ce que l'on appelle des écouteurs qu'elles seront écrites. Il y a trois types d'écouteurs :

- *L'écouteur par défaut :* pris en charge par la classe `DefaultTraceListener`, il permet d'écrire dans la console. Cela vous intéresse moyennement, car c'est équivalent à un simple affichage, comme le font les fonctions de la classe `Console`. Vous perdez une grande partie de l'intérêt des journaux de traces.

- *Les fichiers texte :* c'est l'une des possibilités les plus intéressantes, et celle que nous allons traiter d'ailleurs. Elle est à la fois, simple, pratique et efficace. Il faut pour cela utiliser la classe `TextWriterTraceListener`.
- *Les journaux d'événements :* nous avons par abus de langage utilisé le terme "journaux d'événements" pour parler des traces de manière générale. Mais cette appellation est censée être réservée aux traces dont l'écouteur est un `EventLogTraceListener`. Cela permet de travailler avec un objet `EventLog`, dont les possibilités sont plus étendues qu'un simple fichier texte. Mais dans le cadre de cet ouvrage, cela ne sera pas nécessaire. Le fichier texte permet déjà un grand nombre de possibilités et offre une stabilité importante.

Petite explication sur le Flush

Nous vous en avons brièvement parlé lors de la description des méthodes d'écriture et nous allons vous expliquer pourquoi ce mécanisme est important.

Lorsque des données doivent être écrites, elles sont d'abord mises dans une mémoire tampon. Vous avez certainement déjà entendu le terme de "buffer" ? C'est de cela dont il s'agit. Traiter les éléments de manière individuelle demande trop de temps de calcul, de traitement différents. Lorsque l'on veut afficher du texte à l'écran, il faut ouvrir un accès à l'écran, écrire le texte en question, puis refermer cet accès. Il s'agit là d'un résumé des opérations principales et elles prennent beaucoup de temps. Alors imaginez qu'on écrive les caractères un par un, on multiplie d'autant ces opérations. C'est pourquoi, dans la plupart des cas d'écriture, on écrit tout dans un buffer, dont l'accès est direct et ne demande pas d'opération particulière. Une fois que celui-ci est rempli, ou alors quand on l'a décidé, on écrit le contenu du buffer et on le vide.

Le `Flush` consiste à vider le buffer et à en imprimer le contenu. Il y a plusieurs moyens d'effectuer cette opération. L'impression d'un caractère de retour à la ligne par exemple effectue un `Flush`. C'est pourquoi il est recommandé d'écrire des chaînes de caractères avec les méthodes de type `WriteLine` plutôt que `Write`, qui ne fait pas le `Flush` automatiquement, étant donné qu'elle n'écrit pas de retour à la ligne. Mais il est également possible de le faire par l'appel de fonctions spécifiques, ou la définition de propriétés particulières.

Dans le cas de la création de journaux, on peut utiliser la méthode `Flush` des classes `Trace` et `Debug` après avoir écrit les informations dans le log :

```
Trace.Write("Message à écrire")
Trace.Flush()
```

Bien que l'on n'ait pas inséré de retour à la ligne, le message sera écrit directement et le buffer sera vidé, car on a explicitement fait appel à la fonction `Flush`. Cette méthode est certes efficace, mais aussi contraignante, car elle oblige à faire un appel à la fonction `Flush` chaque fois que l'on écrit un message. Heureusement, les classes `Debug` et `Trace` ont un attribut booléen `AutoFlush` qui permet de réaliser automatiquement l'appel à `Flush` après chaque écriture. Il suffit de le définir à `true` pour que le `Flush` soit automatique et que les messages soient écrits directement à chaque fois.

```
Debug.AutoFlush = true
Debug.Write("Message de Debug 1")
Debug.Write("Message de Debug 2")
```

Attention, `Write` n'écrit pas de caractères supplémentaires après le message, donc la sortie sera la suivante :

```
Message de Debug 1Message de Debug 2
```

Voila l'essentiel du fonctionnement du `Flush` et des buffers. Vous comprenez maintenant l'intérêt d'utiliser les fonctions comme `WriteLine`, et vous savez pourquoi certaines écritures que vous commandez ne sont pas affichées tout de suite.

Utilisation de l'écouteur par défaut

L'écouteur par défaut est celui utilisé quand aucun écouteur n'a été défini précisément. Il correspond à la fenêtre de sortie de Visual Studio. Prenons un exemple :

```
Trace.WriteLine("Début de la première opération")
```

Si vous n'avez pas utilisé du tout `Trace` ou si vous vous êtes juste contenté d'écrire des messages, il prendra l'écouteur par défaut. De ce fait, le message s'affichera dans Visual Studio (dans la mesure où vous avez bien géré les `Flush`, mais ici vous faites un `WriteLine`, donc il n'y a pas de problème).

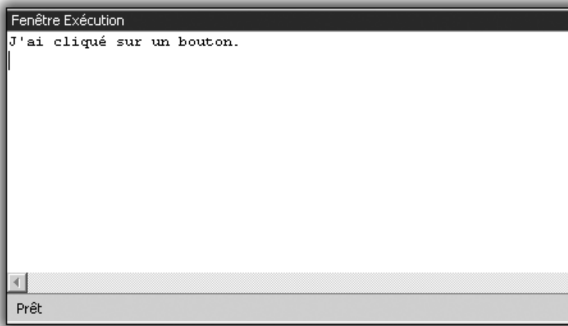


Figure 8.7 : *Affichage de Trace dans l'écouteur par défaut*

Utilisation de l'écouteur pour les fichiers texte

Si vous utilisez cet écouteur particulier, les messages seront écrits dans un fichier texte. Ainsi, vous pourrez lire les traces même si le programme n'est pas exécuté. Cela est intéressant pour diverses raisons. D'une part, lire un fichier texte est plus agréable que de chercher sur une console (la fenêtre **Exécution** de Visual Studio par exemple). D'autre part, cela permet de conserver une trace écrite de ce qu'il s'est passé, même si le programme ou Visual Studio n'est pas en cours d'exécution, et ainsi de pouvoir régler plus facilement des problèmes par rapport à telle ou telle étape.

Pour mettre en œuvre un tel écouteur, procédez ainsi :

1. Tout d'abord, assurez-vous qu'un fichier texte existe pour ces opérations. Vous pouvez soit en créer un, soit en ouvrir un qui existe déjà et, à ce moment-là, décidez si vous préférez l'écraser ou y ajouter les nouveaux messages.
2. Ensuite, créez l'écouteur correspondant. Il suffit d'instancier un objet `TextWriterTraceListener` dont la cible sera le fichier texte précédemment préparé.
3. Enfin, dites aux classes `Debug` et `Trace` que l'écouteur qu'elles doivent utiliser maintenant est le `TextWriterTraceListener` que vous venez de créer. Pour cela, ajoutez cet écouteur à la collection `Listeners` des classes `Debug` et `Trace` (qui représente leurs écouteurs), grâce à la méthode `Add`.

Voici le code qui permet de faire cela :

- Déclaration de la variable correspondant au fichier texte :

```
Dim monLog As System.IO.FileStream
```

- Ouverture du fichier texte. Il est créé s'il n'existe pas. S'il existe, il est écrasé.

```
monLog = New("c:\Projet\monFichierLog.txt", System.IO
✂ .FileMode.OpenOrCreate)
```

Voici la méthode pour ajouter les messages à un fichier existant (faites l'une ou l'autre, mais pas les deux) :

```
monLog = New("c:\Projet\monFichierLog.txt", System.IO
✂ .FileMode.Append)
```

- Déclaration de l'écouteur et association avec le fichier texte :

```
Dim monEcouteur As TextWriterTraceListener
monEcouteur = New TextWriterTraceListener(myLog)
```

- Ajout de l'écouteur à la liste des écouteurs de `Trace` ou `Debug` :

```
Trace.Listeners.Add(monEcouteur)
Debug.Listeners.Add(monEcouteur)
```

Une fois ces diverses opérations effectuées, les traces de `Debug` et `Trace` seront marquées dans le fichier texte spécifié. Attention toutefois à bien gérer les `Flush`, car cela pourrait provoquer des parasites dans les logs.

Nous ne verrons pas ici l'écriture dans les vrais journaux d'événements avec les `EventLog`. Mais avec une bonne maîtrise de l'écriture de messages d'information dans des fichiers texte, vous serez déjà en mesure d'améliorer grandement la qualité et la stabilité de vos applications. L'utilisation de tels fichiers texte dans des applications sensibles ou des projets mondialement connus montre que c'est un système largement utilisé, car efficace. Il ne tient maintenant qu'à vous de les remplir d'informations pertinentes. Date, valeur des variables, utilisateur qui a provoqué l'opération, poste de travail, les possibilités ne manquent pas. À vous de voir celles qui vous seront les plus utiles.

8.3. Ne pas se perdre

Lors de l'élaboration d'un projet, il est courant que les fichiers de code dépassent plusieurs centaines de lignes. Si vous avez cinquante fichiers de plus de cent lignes, il est facile d'oublier à quoi correspond la partie que vous avez écrite. Pour s'y retrouver, il est fortement recommandé d'utiliser les commentaires. Il s'agit de lignes insérées dans les fichiers de code et qui ne seront pas interprétées par le compilateur. Ils sont traditionnellement utilisés pour décrire le fonctionnement des fonctions ou méthodes, ou l'utilité de telle ou telle variable. Pour mettre une ligne en commentaire dans Visual Basic Express, il suffit de la faire précéder d'un apostrophe simple, comme le montre l'exemple suivant :

```
   Sub MaProcedure()
       'Cette procedure sert à _
mettre en valeur les commentaires.
  MessageBox.Show("Bonjour")
  'La ligne précédente affiche Bonjour.
   End Sub
```

Lancez ensuite le programme en faisant appel à cette procédure à travers un bouton ou n'importe quel autre contrôle. Vous remarquerez que la seule ligne à être exécutée par le programme est celle du milieu ; les autres sont ignorées et ne servent que de repère au développeur. Elles se distinguent par leur couleur verte dans l'éditeur de code de Visual Basic.

Chapitre 9

Passer au niveau supérieur

Avec les acquis des chapitres précédents, vous êtes en mesure de créer un grand nombre d'applications et d'affûter vos talents de programmeur. C'est en forgeant qu'on devient forgeron, et c'est par beaucoup de pratique que vous allez progresser et comprendre les subtilités de la programmation. Cependant, il est important d'étoffer votre bagage théorique. Dans ce chapitre, vous allez voir des concepts un peu plus avancés, que vous avez peut-être utilisés sans le savoir.

Jusqu'à présent, vous avez manipulé des objets, des classes, sans forcément savoir ce que cela représentait réellement, ou pourquoi il fallait les utiliser de telle ou telle manière. Dans ce chapitre, nous expliquerons les principes de la programmation orientée objet, ainsi que les possibilités qu'elle offre en termes de programmation, de compréhension ou de sécurité.

Ensuite, nous examinerons la vie des données. Quand apparaissent-elles, où sont-elles stockées, quand disparaissent-elles ?

Le fait de manipuler les objets et tous les mécanismes qui s'y rapportent permet de les structurer de manière intelligente pour former des morceaux de programme dont les fonctionnalités sont intéressantes. Ce sont les modèles de conception, ou encore design patterns. Au cours de ce chapitre, vous en découvrirez quelques-uns et en implémenterez.

La bonne qualité d'un programme passe aussi par les habitudes du programmeur. Jusqu'ici, vous avez utilisé implicitement certaines conventions. Mais il est important de bien les décrire, pour que vous les utilisiez correctement, et surtout, en sachant pourquoi. C'est ce que nous exposerons à la fin de ce chapitre.

9.1. La programmation orientée objet

Dans les chapitres précédents, la plupart des éléments que vous avez utilisés sont des objets ou des classes. Par exemple, vous avez employé les classes `Debug` et `Trace` et manipulé des exceptions, qui sont des objets de la classe `Exception`.



*Reportez-vous à ce sujet au chapitre **Rendre un programme robuste.***

Une question demeure : qu'est-ce qu'un objet ? À vrai dire, la réelle question à se poser est : qu'est ce qu'une classe ? Une classe est un modèle d'objet. Cette réponse n'est pas fausse. Mais si l'on utilise les classes pour définir les objets et que, pour définir les objets, on utilise les classes, on tourne en rond. En réalité, une classe est un concept, c'est-à-dire l'ensemble des éléments qui vont définir quelque chose de concret.

Prenons un exemple, utilisé dans le monde de l'informatique pour décrire ce que sont les objets et les classes : un véhicule. Comment décrire un véhicule ? Par son nombre de roues, son moteur, sa marque, son modèle, son année de mise en circulation, sa puissance, le type mine. Les possibilités sont nombreuses. Une classe est le regroupement de tous les éléments qui décrivent une voiture. C'est un peu comme le tableau des caractéristiques techniques, à ceci près que celui-ci serait vide. Il n'y aurait que les lignes avec les différents critères.

L'objet est le véhicule concret. Le vôtre, celui du voisin, celui garé devant votre portail et qui vous bouche la sortie. Ils ont chacun leurs caractéristiques. Si l'on reprend l'analogie avec le tableau des caractéristiques techniques d'un véhicule, un objet correspondrait à une colonne dans laquelle figureraient les valeurs des critères (les éléments de la classe) qui décrivent ce véhicule.

En programmation, les objets peuvent être des exceptions, des connexions à des bases de données, des boutons, des formulaires, etc. Il s'agit de structures de programmation contenant les données et les traitements nécessaires à leur fonctionnement, autrement appelés "membres".

Création d'une classe et de ses membres

Les membres d'un objet représentent les données et les fonctionnalités inhérentes à cet objet. Il peut y avoir des attributs, des méthodes, des propriétés et des événements. Les attributs et les propriétés représentent le côté "données" de l'objet, alors que les méthodes et les événements représentent le coté "fonctionnalités". Voyons comment créer une classe.

1 Tout d'abord, il faut indiquer que l'on va créer une classe :

```
Public Class Voiture
  ' Tout ce qui ce sera dans cette partie concernera
  ✂ la classe Voiture
End Class
```

Cela délimite dans le programme l'endroit où sera décrite la classe `Voiture`.

2 Ensuite, il faut ajouter quelques attributs qui permettent de la définir.

Pour l'instant, le principe utilisé rappelle un peu celui des enregistrements. Mais le concept d'objet va au-delà, comme vous le verrez au fur et à mesure.

```
Public Class Voiture
  ' Sa marque
  Private marque As String
  ' Son modèle
  Private modele As String
  ' Son année de mise en circulation
  Private anneeCircu As Integer
  ' Sa puissance, en chevaux
  Private puissance As Integer
  ' Le niveau d'essence, en litres
  Private essence As Integer

End Class
```

3 Enfin, il faut ajouter quelques méthodes pour décrire les fonctionnalités.

Les méthodes sont des fonctions appliquées à une classe. Elles peuvent directement utiliser les attributs de la classe pour les traitements. Cependant, pour des raisons de propreté du code source, utilisez plutôt `Me`. `Me` est un mot-clé qui désigne l'objet en cours. Il permet d'accéder à ses attributs et aux méthodes. De plus, si le paramètre de la méthode porte le même nom que l'un des attributs de la classe (ce qui est possible, mais pas recommandé), `Me` permet de faire la distinction.

```
Public Class Voiture
  ' Sa marque
  Private marque As String
  ' Son modèle
  Private modele As String
  ' Son année de mise en circulation
  Private anneeCircu As Integer
  ' Sa puissance, en chevaux
  Private puissance As Integer
  ' Le niveau d'essence, en litres
  Private essence As Integer

' Méthode pour faire le plein
  ' Exemple décrivant l'importance de Me
```

```
  ' Évitez les noms de variable identiques
Public Sub faireLePlein(ByVal essence As Integer)
    Me.essence = Me.essence + essence
End Sub

' Méthode pour tuner le moteur
Public Sub gonflerMoteur(ByVal cvGagnes As Integer)
Me.puissance = Me.puissance + cvGagnes
End Sub

End Class
```

Remarquez la présence des mots-clés `Private` et `Public` devant les attributs et les méthodes. Ils définissent le degré d'encapsulation du membre.

Encapsulation

L'encapsulation est l'un des mécanismes de la programmation orientée objet. Ce principe permet de séparer dans le programme l'interface et l'implémentation. Par "interface", on entend ce qui est visible par le monde extérieur. L'objet sera amené à être en relation avec d'autres objets. Par exemple, un véhicule va être en relation avec un conducteur ou avec un autre véhicule, avec une contravention. De cette manière, on garantit plus ou moins que, tant que cette interface n'est pas modifiée, les interactions avec l'extérieur restent valables. L'implémentation correspond à ce qu'il se passe à l'intérieur, c'est-à-dire le fonctionnement interne de l'objet. Celui-ci n'a pas besoin d'être connu de l'extérieur pour permettre à l'objet de bien fonctionner. Plus que cela, vous n'avez pas forcément envie que le reste du monde puisse accéder à vos secrets de fabrication. Cela est un peu imagé, mais permet d'expliquer les concepts d'interface et d'implémentation. En ce qui concerne la voiture, l'interface serait le fait qu'elle roule. L'implémentation serait le fonctionnement du moteur à l'intérieur. Pour faire rouler la voiture, on n'a pas besoin de savoir comment cela se passe en interne. De plus, les constructeurs n'ont pas intérêt à divulguer les secrets de construction de leurs moteurs. Pour le programme informatique, il en est de même.

Il y a plusieurs degrés d'encapsulation, appelés "modificateurs d'accès" :

- **Privé** : traduit par le mot-clé `Private`, ce modificateur empêche tout élément qui ne fait pas partie de la classe en cours d'accéder au membre. D'un point de vue de la sécurité, il est recommandé

de mettre tous les attributs en privé pour éviter toute modification intempestive.

- **Public** : traduit par le mot-clé `Public`, ce modificateur autorise l'accès au membre de n'importe où et par n'importe qui. En termes de sécurité, il ne convient pas aux attributs, mais pour les méthodes, c'est généralement le modificateur d'accès ad hoc.
- **Protégé** : traduit par le mot-clé `Protected`, ce modificateur autorise les classes qui dérivent de la classe en cours d'accéder au membre. Cela intervient lors de l'héritage, qui est un autre mécanisme important de la programmation objet (nous y reviendrons plus loin).
- **Interne** : traduit par le mot-clé `Friend`, ce modificateur limite l'accès au membre exclusivement aux types du même assemblage. L'assemblage correspond en fait au programme. Son utilisation est subtile et n'entre pas le cadre de cet ouvrage. Mais sachez qu'il existe.
- **Interne protégé** : cumule les possibilités d'accès de `Friend` et de `Protected`. Le membre est accessible soit aux types de l'assemblage, soit aux classes qui héritent de la classe en cours.

Les modificateurs les plus importants et les plus utiles sont `Public` et `Private`. Il sera question de `Protected` lorsque nous aborderons l'héritage. Voyons un peu les différences entre les attributs `Public` et `Private` et comment en tirer le meilleur parti. Voici deux classes et quelques membres :

```
Class Voiture
  ' Attribut année de mise en circulation
  Public anneeCirculation As Integer
End Class

Class Personne
  ' Tentative de modification de l'année de mise en
  ✂ circulation
  Public Sub frauderVoiture(laVoiture As Voiture)
    laVoiture.anneeCirculation = laVoiture.anneeCirculation
    ✂ + 5
  End Sub
End Class
```

La méthode `frauderVoiture` traduit l'intention d'une personne d'augmenter l'année de mise en circulation du véhicule pour simuler le fait que celui-ci est plus récent. D'un point de la programmation, il n'y pas d'erreur : le programme se compilera et s'exécutera sans souci. Il

sera alors possible de modifier la date de mise en circulation d'une voiture sans problème. Pas très sécurisant n'est-ce pas ? C'est pourquoi il est recommandé, dans la mesure du possible, de définir les attributs comme privés, de manière que l'extérieur ne puisse pas provoquer de comportement non autorisés. Essayons de remédier à ce problème :

```
Class Voiture
  ' Marque et modèle
  Private marque As String
  ' Attribut année de mise en circulation
  Private anneeCirculation As Integer
End Class

Class Personne
  ' Tentative de modification de l'année de mise en
  ✂ circulation
  Public Sub frauderVoiture(laVoiture As Voiture)
    laVoiture.anneeCirculation = laVoiture.anneeCirculation
    ✂ + 5
  End Sub

  ' Passer une annonce de vente
  Public Sub vendreVoiture(laVoiture As Voiture)
    Dim texteAnnonce As String
    texteAnnonce = "Vends voiture " & laVoiture.marque
texteAnnonce = texteAnnonce & " immatriculée en "
    texteAnnonce = texteAnnonce & Voiture.anneeCirculation
    MessageBox.Show(textAnnonce)
  End Sub
End Class
```

Le problème précédent est réglé. La méthode `frauderVoiture` ne peut plus fonctionner, car elle n'a plus accès au membre `anneeCirculation`, celui-ci étant maintenant privé. Cela provoquera une erreur à la compilation. On ne peut accéder au membre car celui-ci est privé.

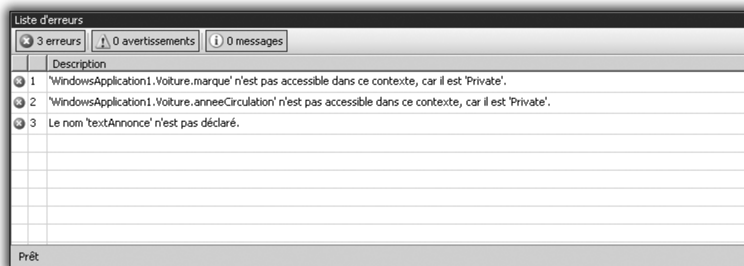


Figure 9.1 : *Erreur d'accès à un membre privé*

La fonction `vendreVoiture` pose un autre problème. En effet, il est impossible de construire le texte de l'annonce, étant donné que la personne n'a pas accès à la marque de la voiture ni à l'année de mise en circulation. Les attributs ne sont plus du tout visibles de l'extérieur, et perdent alors leur intérêt. Pour remédier à ce problème, on utilise des méthodes publiques, qui manipulent les données et qui sont accessibles de l'extérieur. De cette manière, l'accès aux attributs est permis, mais vous gardez un contrôle sur ce que peut faire l'extérieur.

```
Class Voiture
  ' Marque et méthode d'accès
  Private marque As String
  Public Function getMarque() As String
    getMarque = Me.marque
  End Function
  ' Attribut année de mise en circulation
  Private anneeCirculation As Integer
  ' Méthode pour accéder à la date de mise en circulation
  Public Function getAnneeCircu() As Integer
    getAnneeCircu = Me.anneeCircu
  End Function
End Class

Class Personne
  ' Tentative de modification de l'année de mise en
  ✂ circulation
  Public Sub frauderVoiture(laVoiture As Voiture)
    laVoiture.anneeCirculation = laVoiture.anneeCirculation
    ✂ + 5
  End Sub

  ' Passer une annonce de vente
  Public Sub vendreVoiture(laVoiture As Voiture)
    Dim texteAnnonce As String
    texteAnnonce = "Vends voiture " & laVoiture.getMarque()
texteAnnonce = texteAnnonce & " immatriculée en "
    texteAnnonce = texteAnnonce & laVoiture.getAnneeCircu()
    MessageBox.Show(textAnnonce)
  End Sub
End Class
```

Ainsi, les attributs sont bien protégés car ils sont privés. Cependant, on peut les utiliser quand même grâce aux méthodes `getMarque` et `getAnneeCircu`. Ces deux méthodes publiques correspondent à l'interface de la classe `Voiture`, et le reste à son implémentation.

Les accesseurs et propriétés

Les méthodes getMarque et getAnneeCircu sont des cas particuliers d'une catégorie de méthodes appelée "accesseurs". En effet, comme il faut généralement définir les attributs comme privés, on se retrouve bloqué dès que l'on veut y accéder de l'extérieur. C'est pourquoi on crée les méthodes publiques getMarque et getAnneeCircu, pour lire la valeur de l'attribut, sans permettre à l'extérieur d'agir directement dessus. Ces méthodes sont des accesseurs en lecture, aussi appelés "*getter*". Par convention, leur nom commence par get, suivi du nom de l'attribut concerné. L'autre type d'accesseur est l'accesseur en écriture, aussi appelé "*setter*". Par convention, le nom d'une telle méthode commence par set, suivi du nom de l'attribut concerné. On peut alors contrôler les modifications faites sur les attributs :

```
Class Voiture
  ' Marque et méthode d'accès
  Private marque As String
  ' Getter de la marque
  Public Function getMarque() As String
    getMarque = Me.marque
  End Function
  ' Setter de la marque
  Public Sub setMarque(ByVal laMarque As String)
    Me.marque = laMarque
  End Function

  ' Attribut année de mise en circulation
  Private anneeCirculation As Integer
  ' Getter de l'année de mise en circulation
  ' On vérifie qu'elle est valable, non supérieure à
  ✂ l'année en cours
  Public Function getAnneeCircu() As Integer
    If (Me.anneeCircu > 2006)
      MessageBox.Show("Année impossible.")
      getAnneeCircu = -1
    End If
    getAnneeCircu = Me.anneeCircu
  End Function
  ' Setter de l'année de mise en circulation
  ' On vérifie aussi qu'elle est valable
  Public Sub setAnneeCircu(ByVal anCircu As Integer)
    If (anCircu > 2006)
      MessageBox.Show("Année invalide")
    Else
      Me.anneeCircu = anCircu
    End If
  End Sub

End Class
```

Pour changer les valeurs des attributs, il faut faire comme suit :

```
maVoiture.setAnneeCircu(2002)
maVoiture.setMarque("LAMARQUETROPBIEN")
```

Grâce aux accesseurs, vous pouvez manipuler vos attributs avec un maximum de sécurité, tout en les laissant privés pour qu'ils ne soient pas directement manipulables de l'extérieur.

Les propriétés permettent d'automatiser ce principe sans passer par des méthodes spécifiques. Elles permettent de manipuler les attributs, comme les accesseurs, sauf que pour l'extérieur, elles apparaissent comme des attributs (attributs à l'extérieur, méthodes à l'intérieur). Les propriétés combinent la simplicité de l'utilisation directe des attributs, et la sécurité des méthodes, tout en laissant les attributs privés.

Pour illustrer les propriétés, remplaçons les accesseurs de la marque de la voiture par un équivalent avec une propriété :

```
Class Voiture
  ' Marque et méthode d'accès
  Private __marque As String
  ' Propriété pour gérer la marque
Public Property Marque() As String
    ' Accesseur de lecture
    Get
      Return Me.__marque
    End Get
    ' Accesseur d'écriture
    Set(ByVal Value As String)
      Me.__marque = Value
    End Set
End Property
[…]
End Class
```

Détaillons ce bout de code. La définition de la propriété se fait entre `Public Property` et `End Property`. Il faut qu'une propriété soit publique. Sinon on ne peut pas y accéder, ce qui n'a aucun intérêt. Remarquez que l'on a ajouté __ devant le nom de l'attribut. En effet, il ne faut pas que celui-ci ait le même nom que la propriété. Par convention, on donne à la propriété le nom le plus explicite, étant donné que c'est elle qui sera accessible de l'extérieur, et l'on marque le nom de l'attribut par des caractères (généralement des underscores _) au début ou à la fin.

Dans la propriété, on définit les accesseurs. Ils ont le même rôle que les méthodes vues précédemment, `getMarque` et `setMarque`. Leur définition se fait entre `Get` et `End Get` pour l'accesseur en lecture, et entre `Set` et `End Set` pour l'accesseur en écriture. Le `Set` a un paramètre. Son utilisation se fait sous la forme d'une affectation, c'est-à-dire une instruction de la forme `a = b`. Le paramètre `Value` correspond en fait à l'élément à droite du égal (=), la nouvelle valeur.

Il est possible de définir des propriétés en lecture seule ou en écriture seule. Pour la lecture seule, il faut spécifier `ReadOnly` sur l'attribut et la propriété, et n'écrire que l'accesseur `Get`. Pour l'écriture seule, il faut marquer l'attribut et la propriété comme `WriteOnly`, et n'écrire que l'accesseur `Set`.

```
Class Voiture
  ' Marque et méthode d'accès
  Private ReadOnly __marque As String
  ' Propriété en lecture seule pour obtenir la marque
Public Property ReadOnly Marque() As String
    ' Accesseur de lecture
    Get
      Return Me.__marque
    End Get
    ' Pas d'accesseur en écriture
End Property

' Attribut de puissance, écriture seule
Private WriteOnly __puissance As Integer
' Propriété pour la puissance, écriture seule
Public WriteOnly Property Puissance() As Integer
    ' Pas d'accesseur en lecture
    ' Accesseur en écriture
    Set(ByVal Value As Integer)
      Me.__puissance = Value
    End Set
End Property
End Class
```

Le passage en écriture ou lecture simple est simple. Ainsi, l'utilisation des propriétés permet de combiner l'intérêt de passer par des méthodes, c'est-à-dire de pouvoir approfondir le traitement, tout en gardant une simplicité d'utilisation. En effet, les propriétés s'utilisent comme des champs :

```
MessageBox.Show("Je possède une : " & maVoiture.Marque)
maVoiture.Puissance = 150
MessageBox.Show("J'ai tenté l'amélioration _
de ma voiture à 150 CV")
```



Figure 9.2 :
Première MessageBox

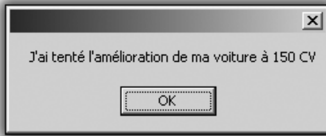


Figure 9.3 :
Seconde MessageBox

La plupart des mécanismes qui ont été expliqués jusque-là concernent les objets, c'est-à-dire des instances des classes que vous avez définies. Cependant, certains éléments sont parfois communs à toutes les instances des classes. C'est pourquoi il est possible de les définir au niveau de la classe, et non plus de l'objet. Ce sera le sujet de la section suivante.

Les membres partagés

Lors de l'utilisation des classes `Debug` et `Trace`, vous n'avez pas employé d'objet de ces classes. Vous appeliez des méthodes directement en écrivant `Debug.WriteLine` ou `Trace.WriteLine`. Cela est possible car ces membres des classes `Debug` et `Trace` sont partagés. En d'autres termes, ils ne sont pas spécifiques à un objet, mais communs à tous les objets de la même classe.

Revenons à l'exemple des voitures. Quelle qu'elle soit, elle possède quatre roues. Il est alors plus judicieux de dire que toutes les voitures ont quatre roues, que de spécifier pour chacune des voitures existantes qu'elle a quatre roues.

Examinons la classe suivante :

```
Public Class Voiture
  Public nbreRoue As Integer
End Class
```

On est obligé de spécifier pour chacune des voitures qu'elle a quatre roues :

```
Dim voiture1 As New Voiture()
Dim voiture2 As New Voiture()
Dim voiture3 As New Voiture()
Voiture1.nbreRoue = 4
Voiture2.nbreRoue = 4
Voiture3.nbreRoue = 4
MessageBox.Show("Nbre de roues la voiture 1 = _
" & voiture1.nbreRoue")
MessageBox.Show ("Nbre de roues la voiture 2 = _
" & voiture2.nbreRoue")
MessageBox.Show ("Nbre de roues la voiture 3 = _
" & voiture3.nbreRoue")
```



Figure 9.4 :
Roues de la voiture 1



Figure 9.5 :
Roues de la voiture 2



Figure 9.6 :
Roues de la voiture 3

Cela est répétitif pour une information que l'on va retrouver tout le temps. Imaginez si le nombre de voitures est particulièrement grand. On ne s'en sort plus. Il est plus judicieux alors de définir le nombre de roues comme un membre partagé par tous les objets de cette classe. Pour cela, on utilise le mot-clé `Shared`.

```
Public Class Voiture
  Public Shared nbreRoue As Integer = 4
End Class
```

De cette manière, on sait d'avance que toutes les voitures qui vont être créées auront un nombre de roues égal à quatre. Le fait que ce soit un membre partagé n'impose plus l'existence d'un objet.

Avant il fallait créer un objet pour un accéder à son membre :

```
Dim maVoiture As New Voiture()
maVoiture.nbreRoue = 4
```

Maintenant ce n'est plus nécessaire. On peut accéder au membre concerné tout simplement en appelant la classe.

```
MessageBox.Show("Une voiture a " & _
Voiture.nbreRoue & " roues.")
```

Le bout de programme précédent s'en retrouve transformé :

```
Dim voiture1 As New Voiture()
Dim voiture2 As New Voiture()
Dim voiture3 As New Voiture()
MessageBox.Show ("Nbre de roues la voiture 1 = _
 " & Voiture.nbreRoue")
MessageBox.Show ("Nbre de roues la voiture 2 = _
 " & Voiture.nbreRoue")
MessageBox.Show ("Nbre de roues la voiture 3 = _
 " & Voiture.nbreRoue")
```

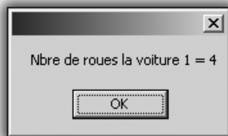


Figure 9.7 : *Roues de la voiture 1, identique aux autres*

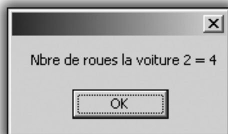


Figure 9.8 : *Roues de la voiture 2, identique aux autres*



Figure 9.9 : *Roues de la voiture 3, identique aux autres*

Il n'est plus nécessaire de donner l'information pour chacune des voitures pour obtenir le même résultat. De plus, si les voitures étaient un jour dotées de six roues au lieu de quatre, il ne serait pas nécessaire d'appliquer une modification à toutes les voitures existantes. Il suffirait de changer le membre partagé et la modification affecterait l'ensemble des voitures existantes :

```
Voiture.nbreRoue = 6
```

Attention, on ne peut pas accéder à un membre partagé dans un contexte non partagé, et vice versa.

```
Public Class Voiture
  Public Shared nbreRoue As Integer = 4
Public marque As String
End Class
```

On provoquerait une erreur en voulant accéder à un membre non partagé en passant par la classe :

```
Voiture.marque = "LAMARQUETROPBIEN"
```

Le compilateur le signalerait.

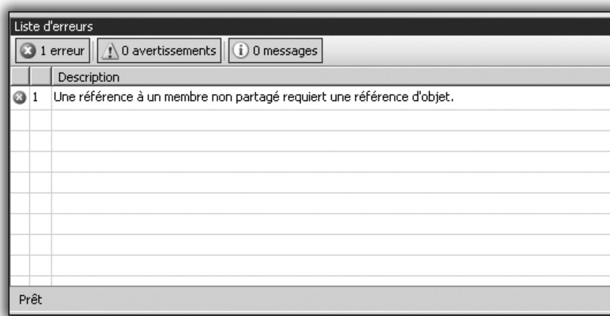


Figure 9.10 : *Erreur d'accès à un attribut non partagé dans un contexte partagé*

C'est également impossible dans l'autre sens :

```
Dim maVoiture As New Voiture
MessageBox.Show(maVoiture.nbreRoue)
```

Cela provoquerait également une erreur mais dans le sens inverse.

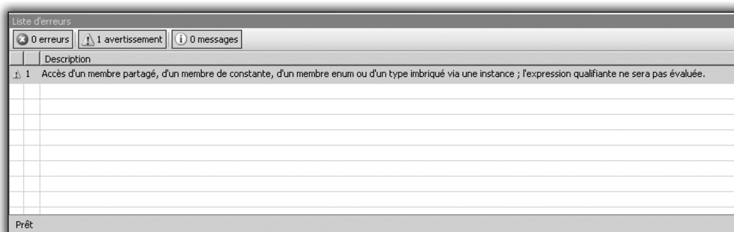


Figure 9.11 : *Erreur d'accès à un attribut partagé dans un contexte non partagé*

Lorsque l'on décrit une classe, il y a toujours des éléments qui sont particuliers à un objet donné et d'autres qui valent pour tous. Ce principe, intéressant et pratique, n'est pas difficile à maîtriser. Il faut juste avoir les idées claires sur ce qui est partagé par tous, et ce qui est spécifique. Ensuite, ce n'est qu'une question d'habitude.

Les méthodes surchargées

La surcharge est un mécanisme pratique de la programmation orientée objet, qui permet d'avoir des membres de même nom. Mais si ces membres ont le même nom, comment les distinguer ? Nous donnerons la réponse dans cette section.

À vrai dire, nous parlons de membres surchargés, mais nous allons surtout nous intéresser au cas particulier des méthodes. En effet, il est possible de créer des méthodes qui ont le même nom, et de les utiliser pour faire des choses différentes. Comment différencier des méthodes qui ont le même nom ? Par leur signature. Qu'est-ce qu'une signature ? C'est la méthode et l'ensemble de ses paramètres.

Prenons ces deux fonctions par exemple :

```
Public Sub afficheAddition(ByVal nbre1 As Integer, ByVal
✂ nbre2 As Integer)
  MessageBox.Show(nbre1 + nbre2)
End Sub

Public Sub afficheAddition(ByVal nbre1 As Integer)
  MessageBox.Show(nbre1 + 3)
End Sub
```

Ces deux fonctions ont le même nom. De plus, elles travaillent sur les mêmes types de variables, à savoir des entiers. Cependant, elles n'ont pas la même signature. En effet, l'une a deux paramètres entiers, alors que l'autre n'en a qu'un seul. Leur signature est différente. Le compilateur et le programme sauront faire la différence lorsqu'un appel sera fait à la première ou à la deuxième méthode.

```
' Appel à la première fonction
afficheAddition(3, 4)
' Appel à la deuxième fonction
afficheAddition(6)
```



Figure 9.12 :
Première fonction



Figure 9.13 :
Seconde fonction

Lorsque vous appelez vos fonctions, le programme analyse le nombre mais également le type des paramètres que vous utilisez. À partir de là, il vérifie s'il connaît des fonctions dont la signature correspond, et il peut alors les exécuter.

Nous avons donné un exemple par rapport au nombre de paramètres. Leur type est tout aussi important :

```
Public Sub afficheAddition(ByVal nbre1 As Integer, ByVal
✂ nbre2 As Integer)
  MessageBox.Show(nbre1 + nbre2)
End Sub

Public Sub afficheAddition(ByVal nbre1 As Double, ByVal
✂ nbre2 As Double)
  MessageBox.Show(nbre1 + 3)
End Sub
```

Vous avez ici les deux mêmes fonctions, sauf que l'une travaille sur des entiers, et l'autre sur des nombres décimaux. Le programme saura également faire la distinction selon les types des paramètres.

```
' Appel à la deuxième fonction
afficheAddition(4.25, 14.3685)
' Appel à la première fonction
afficheAddition(47, 857)
```



Figure 9.14 :
Première fonction



Figure 9.15 :
Seconde fonction

En ce qui concerne l'écriture, les méthodes surchargées sont tout simplement des méthodes et elles respectent les mêmes règles. Vous écrivez la classe normalement, avec ses attributs et ses méthodes, sauf qu'il y aura des moments où certaines auront le même nom :

```
Public Class Additionneur
  Private retenue As Integer
  Private Function effectueAddition(ByVal nb1 As Integer, _
ByVal nb2 As Integer) As Integer
    effectueAddition = nb1 + nb2
  End Function
  Private Function effectueAddition(ByVal nb1 As Double, _
ByVal nb2 As Double) As Double
    effectueAddition = nb1 + nb2
  End Function
End Class
```

Le polymorphisme

Dans la programmation orientée objet, le polymorphisme est sans conteste le mécanisme le plus subtil. C'est également le plus puissant. Il permet de donner au programme une cohérence et une souplesse, et ce de manière propre et intelligente.

Le polymorphisme permet de mettre en relation des classes différentes, de manière qu'elles puissent interagir entre elles, tout en garantissant que les opérations effectuées ont un sens et qu'elles ne vont pas provoquer des résultats impossibles ou totalement absurdes.

Il existe deux types de polymorphismes :

- **Le polymorphisme d'interface** : une interface est un contrat de comportement. Elle présente un ensemble de membres qu'une classe devra définir et implémenter pour pouvoir être utilisé dans ce cadre spécifique. Par exemple, une interface `Transporteur` spécifiera que pour pouvoir être utilisé comme un véhicule transporteur, un véhicule devra implémenter des méthodes de prise de passagers, de desserte de passagers, et de transfert d'un

point à un autre. Le véhicule qui devra être considéré comme un transporteur inclura dans son implémentation l'ensemble de ces comportements. Il aura alors accès aux différents membres qui concernent les transporteurs.

- **Le polymorphisme d'héritage** : l'héritage donne à une classe un certain nombre de fonctionnalités provenant d'une autre classe. On dit alors que la première est une sous-classe de la deuxième, une classe héritée. Elle peut alors utiliser les membres de la classe d'origine sans avoir à les réimplémenter ni à les récrire. Vous avez là un cas de spécialisation. Par exemple, considérez une classe `Losange`. Une classe `Carre` pourrait hériter de `Losange`, car c'est une spécialisation de celle-ci. En effet, un carré est un losange, sauf qu'il possède quatre angles droits. Il n'y aurait alors pas à redéfinir les quatre côtés d'un carré, étant donné qu'ils existent dans un losange.

Voyons cela plus en détail.

Le polymorphisme d'interface

Comme nous l'avons dit, une interface est un contrat de comportement. C'est en quelque sorte une garantie que si une classe implémente comme il faut cette interface, elle aura le comportement adéquat lorsqu'on l'utilisera dans les bonnes conditions.

Une interface se contente de donner la liste des membres à implémenter. Elle n'impose aucune contrainte sur la manière dont est faite cette implémentation.

De plus, il n'y a pas de limite au nombre de classes qui l'implémente. L'interface définit juste un comportement. Elle n'agit pas elle-même. Elle permet juste de mettre en relation différents éléments. Il peut y avoir plusieurs classes qui implémentent une interface, une seule, voire aucune.

En revanche, il n'y a pas de limite au nombre d'interfaces qu'une classe implémente. Par exemple, un bateau peut avoir à la fois un comportement de transporteur de passagers et un comportement de véhicule de course s'il est puissant et léger.

Il y a trois grandes étapes dans l'utilisation d'un polymorphisme d'interface :

- Tout d'abord, il faut définir l'interface, c'est-à-dire l'ensemble des membres que devront implémenter les classes pour être considéré en tant que tel.
- Ensuite, il faut implémenter ces membres dans les classes correspondantes.
- Dès lors, on peut utiliser l'interface comme un intermédiaire entre les différents membres à mettre en relation, sans se soucier de l'implémentation. En effet, étant donné qu'ils implémentent bien l'interface en question, on peut être sûr de leur comportement.

Première étape : la définition de l'interface

Une interface se définit avec le mot-clé `Interface`. Par convention, le nom d'une interface commence toujours par un `I` majuscule. À l'intérieur, il faut mettre les membres qui composeront cette interface. Ce sont généralement des méthodes qui traduisent le comportement. Cependant, on ne peut pas mettre des attributs. Cela empêche qu'un élément mis en relation avec une classe qui implémente cette interface puisse agir directement sur les données internes de celle-ci.

Pour les méthodes, il suffit juste d'en donner le nom et la signature. Par contre, on peut mettre des propriétés, à définir en lecture seule ou en écriture seule.

Pour illustrer ce principe, examinons le problème de transport de passagers dans différents véhicules. Le transport est possible en avion, en voiture, en train, etc. Vous allez étudier comment cela se passe au niveau de chaque élément.

```
Public Class Personne
  Public nom As String
End Class

Public Interface ITransporteur
  Sub prendPassager(passager As Personne)
  Sub deposePassager(passager As Personne)
  Sub allerVers(ByVal destination As String)
End Interface
```

La première partie n'est pas importante. Elle définit simplement une personne représentant le passager que le véhicule devra transporter.

Ensuite, l'interface `ITransporteur` est définie. Maintenant, pour qu'un véhicule soit considéré comme un transporteur potentiel de passagers, il faudra que la classe implémente une méthode

`prendPassager`, ayant pour paramètre une personne, une méthode `deposePassager`, avec le même paramètre, et enfin une méthode `allerVers` dont le paramètre est une chaîne de caractères représentant la ville de destination.

Cette partie est la plus simple dans l'utilisation du polymorphisme d'interface. Voyons maintenant ce qu'il faut faire au niveau des classes qui veulent l'implémenter.

Deuxième étape : l'implémentation de l'interface dans les classes

Considérons deux classes de véhicules, les avions et les bateaux, avec chacune leur comportement de base :

```
Public Class Avion
  Private altitude As Integer
  Private destination As String

  Public Sub decoller()
    Me.altitude = 10000
  End Sub

  Public Sub atterrir()
    Me.altitude = 0
  End Sub

  Public Sub changerCap(ByVal ville As String)
    Me.destination = ville
End Sub
End Class

Public Class Bateau
  Private destination As String

  Public Sub naviguer(ByVal ville As String)
  End Sub
End Class
```

Vous avez l'implémentation de deux classes de véhicules, ayant chacune leurs spécificités. En effet, l'avion doit gérer son altitude, contrairement au bateau qui ne bouge pas en hauteur. Au niveau du bateau, on dit qu'il navigue, alors que l'avion change de cap.

Pour l'instant, aucune d'entre elles n'implémente l'interface définie précédemment et aucune n'est donc en mesure d'être en relation avec des éléments relatifs aux transporteurs de personnes.

L'implémentation de l'interface se passe en deux étapes :

- D'abord, on précise que la classe implémente l'interface.
- Puis, il faut créer une implémentation de chaque membre de l'interface dans la classe.

La partie relative à l'interface elle-même est simple. Il suffit d'utiliser le mot-clé `Implements` avec le nom de l'interface :

```
Public Class Avion
  Implements ITransporteur
  Private altitude As Integer
  Private destination As String

  Public Sub decoller()
    Me.altitude = 10000
  End Sub

  Public Sub atterrir()
    Me.altitude = 0
  End Sub

  Public Sub changerCap(ByVal ville As String)
    Me.destination = ville
End Sub
End Class

Public Class Bateau
  Implements ITransporteur
  Private destination As String

  Public Sub naviguer(ByVal ville As String)
    Me.destination = ville
  End Sub
End Class
```

Si la classe implémente plusieurs interfaces, celles-ci sont séparées par des virgules (on considère que les autres interfaces existent, même si l'on ne les utilise pas ici).

```
Public Class Avion
  Implements Itransporteur, IObjetVolant
  Private altitude As Integer
[…]
End Class

Public Class Bateau
  Implements Itransporteur, IObjetFlottant
  Private destination As String
[…]
End Class
```

Maintenant que vous avez spécifié que les classes devaient implémenter l'interface `ITransporteur`, il faut implémenter chacun des membres, à savoir les méthodes `prendsPassager`, `deposePassager`, `allerVers`.

Pour cela, on utilise également le mot-clé `Implements`, et l'on précise quel membre de l'interface le membre créé implémente. À part cela, il s'agit d'une méthode classique. Attention, pour garantir un bon comportement, il peut être nécessaire de modifier le comportement du reste de la classe. Ici, on ajoutera un attribut `passager` de type `Personne` pour garder un comportement valable.

```
Public Class Avion
  Implements ITransporteur
Private passagerABord As Personne
  Private altitude As Integer
  Private destination As String

  Public Sub decoller()
    Me.altitude = 10000
  End Sub

  Public Sub atterrir()
    Me.altitude = 0
  End Sub

  Public Sub changerCap(ByVal ville As String)
    Me.destination = ville
End Sub

  Public Sub prendsPassager(passager As Personne) _
    Implements ITransporteur.prendsPassager
Me. passagerABord = passager
  End Sub
  Public Sub deposePassager(passager As Personne) _
    Implements ITransporteur. deposePassager
If (Me.passagerABord.nom = passager.nom)
      Me.passagerABord = ""
EndIf
  End Sub
  Public Sub allerVers(ByVal ville As String) _
    Implements ITransporteur.allerVers
    Me.decoller()
    Me.changerCap(ville)
    Me.atterrir()
  End Sub
End Class
```

```
Public Class Bateau
  Implements Itransporteur
Private passagerABord As Personne
  Private destination As String

  Public Sub naviguer(ByVal ville As String)
    Me.destination = ville
  End Sub

  Public Sub prendsPassager(passager As Personne) _
    Implements ITransporteur.prendsPassager
Me. passagerABord = passager
  End Sub
  Public Sub deposePassager(passager As Personne) _
    Implements ITransporteur. deposePassager
    If (Me.passagerABord.nom = passager.nom)
      Me.passagerABord = ""
    EndIf
  End Sub
  Public Sub allerVers(ByVal ville As String) _
    Implements ITransporteur.allerVers
    Me.naviguer(ville)
  End Sub
End Class
```

Maintenant, les deux classes `Bateau` et `Avion` implémentent totalement l'interface `ITransporteur`. Les implémentations se ressemblent. En ce qui concerne le passager, le code est identique. Concernant le transport vers la destination, la différence est que l'avion doit décoller et atterrir. Cela n'a pas d'importance. Comme nous l'avons précisé, l'interface force juste un comportement et n'impose aucune contrainte sur sa "réalisation". Les réalisations auraient pu être identiques ou différentes, cela ne remet pas en question le fait que maintenant, `Avion` et `Bateau` sont considérés comme des `ITransporteur`.

Dernière étape : l'utilisation de l'interface comme intermédiaire

Maintenant que les classes `Avion` et `Bateau` ont implémenté l'interface `ITransporteur`, elles peuvent dialoguer directement avec n'importe quel élément qui attendrait un `ITransporteur`.

Prenons par exemple une fonction `voyage`, qui emmène une personne à un endroit donné, grâce à un transporteur :

```
Public Sub voyage(voyageur As Personne, _
ByVal transport As ITransporteur, _
ByVal ville As String)
End Sub
```

Étant donné que l'on connaît le comportement de toute classe qui implémente l'interface `ITransporteur`, on peut utiliser cela pour définir la fonction `voyage` :

```
Public Sub voyage(voyageur As Personne, _
 ByVal transport As ITransporteur, _
 ByVal ville As String)
  Transport.prendsPassager(voyageur)
  Transport.allerVers(ville)
  Transport.deposePassager(voyageur)
End Sub
```

Pour pouvoir utiliser `Avion` et `Bateau` en tant que `ITransporteur`, il y a une petite opération supplémentaire à réaliser. Il faut convertir explicitement les objets en `ITransporteur`. Pour faire cela, on utilise la fonction de conversion `CType` :

```
Dim monAvion As New Avion()
Dim monBateau As New Bateau()
Dim monAvionTransporteur As ITransporteur
Dim monBateauTransporteur As ITransporteur
monBateauTransporteur = CType(ITransporteur, monBateau)
monAvionTransporteur = CType(ITransporteur, monAvion)
```

Vous avez accompli toutes les étapes pour pouvoir utiliser `Avion` et `Bateau` comme des transporteurs de personnes. Maman veut aller à New York en avion :

```
Dim maman As Personne()
Dim monAvion As Avion()
Dim transporteur As ITransporteur
Dim ville As String
ville = "New York"
maman.nom = "Maman"
transporteur = CType(ITransporteur, monAvion)
voyage(maman, transporteur, ville)
```

Si elle avait voulu partir en bateau, rien de plus simple :

```
Dim maman As Personne()
Dim monBateau As Bateau()
Dim transporteur As ITransporteur
Dim ville As String
ville = "New York"
maman.nom = "Maman"
transporteur = CType(ITransporteur, monBateau)
```

```
voyage(maman, transporteur, ville)
```

Seules deux lignes ont été modifiées, et pourtant le moyen a totalement changé. Vous avez modifié deux appels, tout en garantissant une cohérence et une fiabilité des opérations effectuées. Cet exemple montre l'intérêt du polymorphisme d'interface. C'est un mécanisme subtil et bien pratique une fois maîtrisé.

Le polymorphisme d'héritage

Le polymorphisme d'héritage est le deuxième type de polymorphisme. Il permet de donner à une classe certaines caractéristiques et fonctionnalités d'une classe existante. En quelque sorte, l'héritage permet de spécialiser une classe de base.

La classe spécialisée est dite "*dérivée*" ou "*étendue*", ou encore appelée "*classe fille*".

La classe de base est dite "*étendue*", ou encore appelée "*classe mère*".

Par exemple, prenons une classe de base `Avion`. Un exemple d'héritage possible serait une classe `AvionDeLigne` ou encore `AvionDeChasse`, qui posséderaient les caractéristiques d'un avion, mais qui auraient leurs propres spécificités, par exemple une liste de passagers pour l'avion de ligne, et une liste d'équipements militaires pour l'avion de chasse. Nous utiliserons cet exemple pour illustrer ce mécanisme important de la programmation orientée objet.

Pour indiquer qu'une classe dérive d'un autre classe, il suffit d'utiliser le mot-clé `Inherits`, suivi de la classe de base :

```
Public Class Avion
  ' Nous ne remplissons pas le contenu
  ' Nous le ferons au fur et à mesure
  'de la découverte de l'héritage
End Class

Public Class AvionDeChasse
  Inherits Avion
End Class
```

De cette manière, vous avez fait de la classe `AvionDeChasse` une spécialisation de la classe `Avion`. Contrairement au polymorphisme d'interface, où une classe pouvait implémenter plusieurs interfaces, une

classe ne peut dériver que d'une seule classe de base. Lorsque les deux types de polymorphismes sont utilisés, il faut d'abord écrire l'héritage :

```
Public Class AvionDeChasse
  Inherits Avion
  Impliments ISupersonic
End Class
```

Il est possible d'indiquer qu'une classe n'est pas dérivable quand l'héritage n'apporte aucune information supplémentaire pertinente. Dériver une classe `Cloture` présente peu d'intérêt : si vous avez des clôtures en bois ou des haies, vous pouvez le préciser dans la classe `Cloture` par un attribut de type énumération par exemple. Les fonctionnalités d'une clôture ne changent pas pour autant.

Pour empêcher la dérivation, on utilise le mot-clé `NotInheritable` :

```
Public NotInheritable Class Cloture
  ' Il n'est pas nécessaire de détailler cette classe
End Class

Public Class ClotureHaie
  Inherits Cloture
End Class
```

La classe `Cloture` n'étant pas dérivable, `ClotureHaie` ne pourra pas en hériter. Toute tentative de dérivation provoquera une erreur.

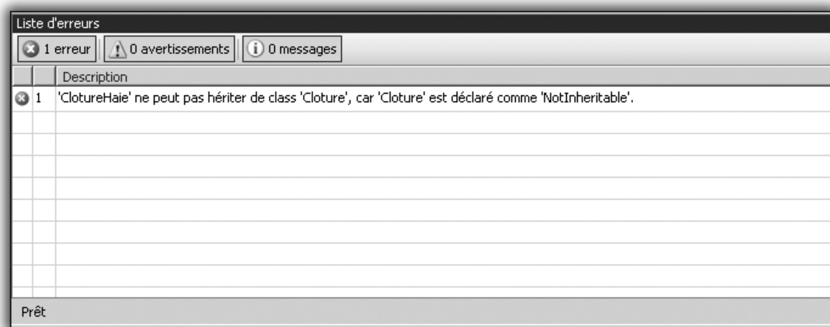


Figure 9.16 : *Tentative de dérivation d'une classe non dérivable*

Vous avez vu comment gérer les héritages au niveau des classes, mais cela ne suffit pas. Il faut aussi le faire au niveau des membres.

L'héritage des membres

Le fait de définir des héritages au niveau des classes permet juste de dire qu'une classe est une classe fille d'une autre. Cependant, une classe fille ne va pas forcément tout hériter de la classe mère. Les membres dont va hériter la classe fille dépendent en grande partie du niveau d'accès et de l'encapsulation de ces membres dans la classe mère. Si le membre de la classe mère est privé, il ne sera pas accessible, même à une classe fille. S'il est public, il est accessible à tout le monde, donc à la classe fille aussi.

```
Public Class Avion
  Private avionDeBase As Boolean
  Public nbreAilerons As Integer
End Class

Public Class AvionDeChasse
  Inherits Avion
  Public Sub afficheMesMembresHerites
    MessageBox.Show(nbreAilerons)
    MessageBox.Show(avionDeBase)
  End Sub
End Class
```

Cela provoquera une erreur car, l'attribut `avionDeBase` étant `Private`, il n'est pas accessible dans la classe `AvionDeChasse`, même si celle-ci dérive de la classe `Avion`.

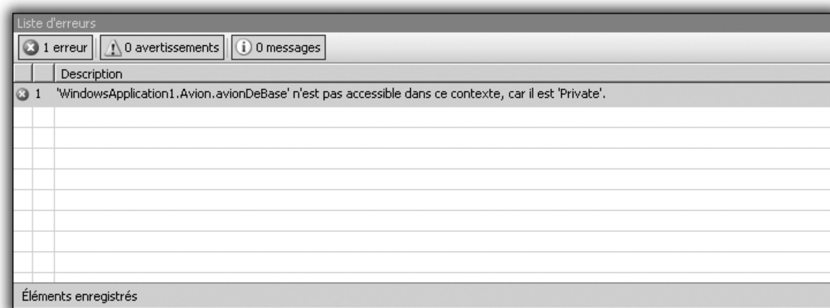


Figure 9.17 : *Tentative d'accès à un membre privé de la classe mère*

Lorsqu'il a été question de l'encapsulation, nous avons parlé du modificateur d'accès `Protected`. Il permet de limiter l'accès à un membre exclusivement à ses classes filles. C'est typiquement le modificateur d'accès utilisé dans les cas d'héritage. Il permet de conserver l'aspect sécuritaire en évitant un accès public. Grâce à lui, les

classes filles peuvent bénéficier de la fonctionnalité correspondante. Dans le code précédent, l'attribut `nbreAilerons` est `Public`, ce qui est déconseillé.

```
Public Class Avion
  Protected vitesseMax As Integer
  Protected nbreAilerons As Integer
  Protected reserveKerozene As Integer
  Public Sub vitesseMaximum()
    Me.vitesseMax = 1000
    MessageBox.Show(Me.vitesseMaximum)
  End
End Class

Public Class AvionDeChasse
  Inherits Avion
  Public Sub afficheMesMembresHerites
    MessageBox.Show(Me.nbreAilerons)
    MessageBox.Show(Me.vitesseMaximum)
    MessageBox.Show(Me.reserveKerozene)
  End Sub
End Class
```

Les membres `reserveKerozene` et `nbreAilerons` étant maintenant protégés, la classe `AvionDeChasse` peut les utiliser directement. Il n'y a alors plus d'erreur sur la méthode `afficheMesMembresHerites`.

La fonction `vitesseMaximum` pousse l'avion à sa vitesse maximale. Comme elle est `Public`, `AvionDeChasse` en hérite et on peut l'appeler :

```
Dim monChasseur As New AvionDeChasse
monChasseur.vitesseMaximum()
```

Cela ne pose pas de problème en termes de programmation. Le compilateur ne détecte pas d'erreur. À l'exécution, l'attribut `vitesseMax` a une valeur de 1 000 et l'affiche dans une boîte de dialogue.

C'est au niveau du sens qu'il y a un problème. Bien qu'étant un avion, un chasseur a une vitesse maximale bien supérieure à celle d'un avion de base, d'un avion de ligne ou d'un planeur. De plus, pour pouvoir atteindre celle-ci, l'avion de chasse doit activer le mécanisme de post-combustion. C'est pourquoi l'implémentation de la vitesse maximale dans la classe de base ne convient plus du tout pour la classe dérivée. Il faut donc la remplacer.

Pour cela, il faut d’abord spécifier dans la classe de base que ce membre est redéfinissable. On utilise en ce sens le mot-clé `Overridable`. Puis dans la classe dérivée, il faut redéfinir et réimplémenter cette méthode. Il faut préciser que la nouvelle méthode est une redéfinition de celle qui existait dans la classe de base. Cela se fait grâce au mot-clé `Overrides`.

```
Public Class Avion
  Protected vitesseMax As Integer
  Protected nbreAilerons As Integer
  Protected reserveKerozene As Integer

  ’ On définit cette méthode comme Overridable
  ’ On pourra alors la redéfinir dans les classes filles
  Public Overridable Sub vitesseMaximum()
    Me.vitesseMax = 1000
    MessageBox.Show(Me.vitesseMaximum)
  End
End Class

Public Class AvionDeChasse
  Inherits Avion
  ’ Activation de la post-combustion
  Public Sub postCombustion()
  ’ L’implémentation n’est pas nécessaire
  ’ pour la compréhension
  End Sub

  ’ Redéfinition de la méthode vitesseMaximum
  ’ Qui est une méthode Overridable de la classe de base
Public Overrides Sub vitesseMaximum()
  Me.postCombustion()
  Me.vitesseMax = 2000
  MessageBox.Show(Me.vitesseMax)
End Sub

  Public Sub afficheMesMembresHerites
    MessageBox.Show(Me.nbreAilerons)
    MessageBox.Show(Me.vitesseMaximum)
    MessageBox.Show(Me.reserveKerozene)
  End Sub
End Class
```

Cela vous permet de redéfinir un comportement dans la mesure où celui-ci est équivalent, c’est-à-dire si la méthode en question a la même signature. Si par exemple, il fallait préciser un paramètre pour la post-combustion, il ne serait plus possible de redéfinir ainsi la méthode `vitesseMaximum`.

Dans ce cas, il est préférable de masquer la méthode de la classe de base. De cette manière, on peut écrire une méthode qui a le même nom, mais dont la signature est différente. Ce principe se rapproche un peu de la surcharge, mais il intervient entre des classes mères et des classes filles, et non pas sur une unique classe. Du côté de la classe mère, rien de ne change. On ne peut masquer que des méthodes `Overridable`. Cependant, au niveau de la classe fille, on utilisera le mot-clé `Shadows`, puis l'on pourra redéfinir normalement la méthode, avec éventuellement une nouvelle signature, une valeur de retour différente, etc.

```
Public Class Avion
  Protected vitesseMax As Integer
  Protected nbreAilerons As Integer
  Protected reserveKerozene As Integer

  ' La méthode vitesseMaximum, redéfinissable
  Public Overridable Sub vitesseMaximum()
    Me.vitesseMax = 1000
    MessageBox.Show(Me.vitesseMaximum)
  End
End Class

Public Class AvionDeChasse
  Inherits Avion
  ' Activation de la post-combustion
  Public Sub postCombustion(ByVal force As Integer)
  ' L'implémentation n'est pas
  ' nécessaire pour la compréhension
  End Sub

  ' Masquage de la méthode vitesseMaximum
  ' La signature est différente de celle de la classe mère
  ' Il y a un paramètre
Public Shadows Sub vitesseMaximum(ByVal force As Integer)
  Me.postCombustion(force)
  Me.vitesseMax = 2000
  MessageBox.Show(Me.vitesseMax)
End Sub
  […]
End Class
```

Bien que cela vous permette de vous adapter au fonctionnement des différentes classes, il y a un risque de problème :

```
Dim monAvion As New Avion()
Dim monChasseur As New AvionDeChasse()
monAvion.vitesseMaximum()
monChasseur.vitesseMaximum()
```

Cela provoquera une erreur, car la fonction de base étant masquée par la nouvelle, `vitesseMaximum` sans paramètre n'existe pas pour la classe `AvionDeChasse`.

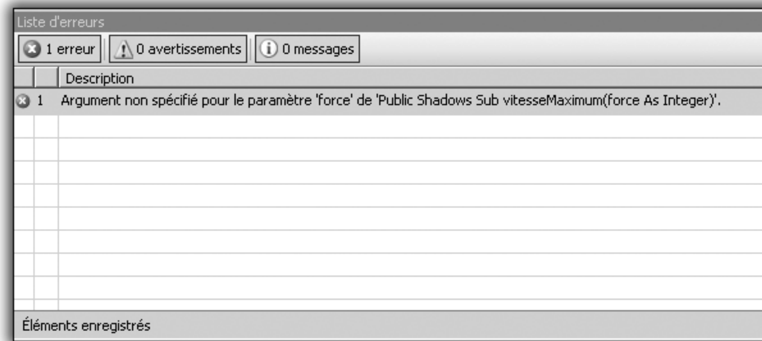


Figure 9.18 : *Tentative d'accès à une fonction masquée*

Il faut donc faire attention lorsque vous utilisez des méthodes masquées, car ce qui était valable à un moment ne l'est plus du tout à un autre.

Le fait de masquer ou de redéfinir une méthode redéfinissable vous donne accès à deux implémentations de cette méthode. Celle qui est utilisée est généralement celle de la classe fille, la plus basse dans la hiérarchie. Cependant, vous pouvez accéder à l'implémentation de la classe mère. Le mot-clé `MyBase` est une référence à la classe mère et permet d'appeler les membres de la classe de base :

```
Public Class AvionDeChasse
  Inherits Avion
  […]
  Public Sub vitesseMaximumMere()
    MyBase.vitesseMaximum()
End Sub
End Class
```

La méthode `vitesseMaximumMere` exécutera la méthode `vitesseMaximum` dans la classe de base, c'est-à-dire qu'elle définira l'attribut `vitesseMax` à 1 000 et l'affichera dans une boîte de dialogue.

L'abstraction

Selon les composants que vous avez à décrire et que vous devez représenter, vous pouvez avoir besoin d'une classe de base qui décrit

une multitude de caractéristiques communes, sans qu'un objet de cette classe puisse exister. Cela veut dire qu'une spécialisation est obligatoire.

Prenons les avions par exemple. La classe de base décrirait qu'un avion a des ailes, une vitesse maximale, un poids, une longueur, etc. Cependant, on peut considérer le concept d'avion trop général et avoir envie de forcer l'utilisation d'une spécialisation pour définir les avions de chasse, les planeurs, les avions de ligne.

Ce mécanisme est l'abstraction. Il empêche la création d'objets d'une classe de base et force donc la dérivation. L'objet dérivé est considéré comme une fille, avec toutes les possibilités vues précédemment.

Pour qu'une classe soit abstraite, il faut utiliser le mot-clé `MustInherit` :

```
Public MustInherit Avion
' L'implémentation n'est pas nécessaire pour la
✂ compréhension
End Class
```

Vous ne pouvez pas créer un avion de base car la classe est abstraite, sous peine d'obtenir une erreur.

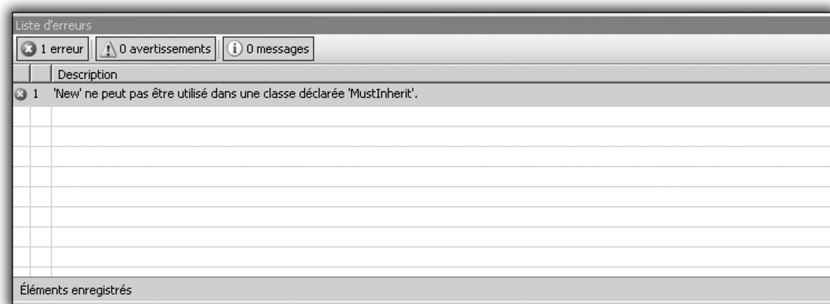


Figure 9.19 : *Tentative d'instanciation d'une classe abstraite*

L'abstraction de la classe vous force donc à spécialiser cette classe. Une fois cela fait, la classe fille aura accès aux membres de la classe de base, comme dans un héritage classique.

Dans une classe abstraite, il est possible de définir des membres abstraits. Cela se rapproche des interfaces. Dans les interfaces, il faut que la classe implémente l'ensemble des membres de l'interface. Pour les membres abstraits, le principe est le même. Pour qu'une classe

puisse dériver d'une classe abstraite qui possède des membres abstraits, elle doit implémenter chacun de ces membres. Dans la classe abstraite, cela se précise par le mot-clé `MustOverride` :

```
Public MustInherit Avion
' L'implémentation n'est pas nécessaire pour la
✂ compréhension
Public MustOverride vitesseMaximum()
End Class
```

Pour pouvoir hériter de la classe `Avion`, une classe devra fournir une implémentation de `vitesseMaximum` :

```
Public Class AvionDeChasse
  Inherits Avion
  ' Activation de la post-combustion
  Public Sub postCombustion(ByVal force As Integer)
  ' L'implémentation n'est pas nécessaire pour la
  ✂ compréhension
  End Sub
' On omet de redéfinir la méthode vitesseMaximum
End Class
```

Comme ce n'est pas le cas ici, l'héritage ne pourra se faire et vous obtiendrez une erreur.

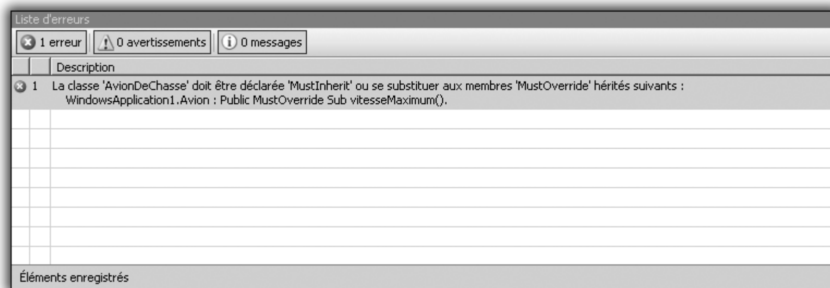


Figure 9.20 : *Instanciation d'une classe fille sans redéfinir les membres abstraits*

Comme pour les membres d'une interface, l'ensemble des membres abstraits doivent avoir une implémentation dans la classe fille :

```
Public Class AvionDeChasse
  Inherits Avion
  ' Activation de la post-combustion
  Public Sub postCombustion()
  'L'implémentation n'est pas nécessaire
  'pour la compréhension
  End Sub
```

```
  'Implémentation de la méthode abstraite
  'de la classe Avion
Public Override Sub vitesseMaximum()
  Me.postCombustion()
  Me.vitesseMax = 2000
  MessageBox.Show(Me.vitesseMax)
End Sub
  […]
End Class
```

Maintenant `AvionDeChasse` peut dériver de la classe `Avion`, qui est abstraite. De cette manière, `AvionDeChasse` bénéficie de toutes les caractéristiques de la classe `Avion`. Mais le fait que celle-ci soit abstraite garantit qu'il ne peut y avoir que des objets complets en termes d'informations et de fonctionnalités.

Vous venez de voir les grandes principes de la programmation orientée objet. Spécialement en Visual Basic .NET, la programmation orientée objet (POO, à prononcer "*pou*") est la base de tout. C'est pourquoi il est important de bien la comprendre et de bien la maîtriser. Le tout est de s'entraîner et de pratiquer.

9.2. La vie des données

En programmation, tous les traitements et la gestion des données sont pris en charge par des mécanismes qui travaillent sur la mémoire en arrière-plan. Les variables correspondent à des emplacements en mémoire avec des valeurs particulières. De la même manière, les objets sont des morceaux de mémoire d'une taille donnée où sont stockées les valeurs nécessaires.

C'est pourquoi la compréhension du fonctionnement de la mémoire permet de mieux comprendre les comportements intervenant dans un programme. Dans cette section, nous allons donner une explication détaillée de ces mécanismes. Vous verrez comment cela se passe au niveau des variables locales, mais aussi des objets, de leurs constructeurs et destructeurs, et comment est récupérée la mémoire vacante grâce au principe du ramasse-miettes (*garbage collector*).

Gestion des variables locales

Vous avez déjà vu au travers de cet ouvrage des utilisations de variables locales. Il s'agit de variables qui sont accessibles et utilisables dans le bloc où elles sont déclarées.

```
Dim X As Integer
For X = 1 to 10
  Dim Y As Integer
  Y = 0
  MessageBox.Show("X = " & X)
  MessageBox.Show("Y = " & Y)
  Next X
MessageBox.Show("X final = " & X)
MessageBox.Show("Y final = " & Y)
```

Dans ce bout de programme, la boucle devrait faire dix tours et afficher les valeurs de `X` entre 1 et 10, et dix fois `Y = 0`. Enfin une boîte de dialogue devrait afficher `X = 10` et `Y = 0`. Or, ce bout de programme provoque une erreur.

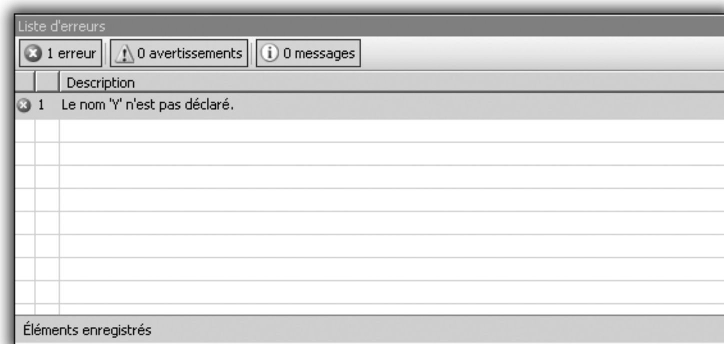


Figure 9.21 : *Accès à une variable hors de portée*

En effet, lors de l'affichage final de `Y`, la variable n'est plus accessible. Étant définie dans la boucle, `Y` n'est accessible que dans cette boucle. Sa portée, c'est-à-dire son accessibilité, est limitée à ladite boucle. On dit que `Y` est une variable locale de cette boucle.

Les variables locales sont gérées dans la pile. La pile est une zone mémoire dédiée à l'utilisation des variables locales. On peut la symboliser comme une vraie pile, composée de plusieurs petits blocs de mémoire que sont les variables :

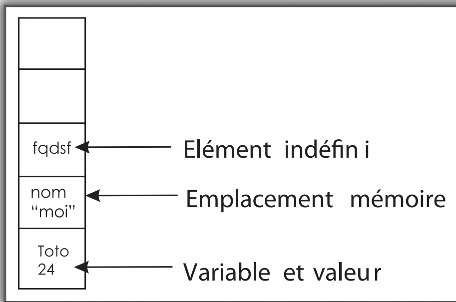


Figure 9.22 : *Représentation simplifiée de la pile mémoire*

Dans cette pile, ce qui va marquer l'accessibilité des variables est un petit marqueur :

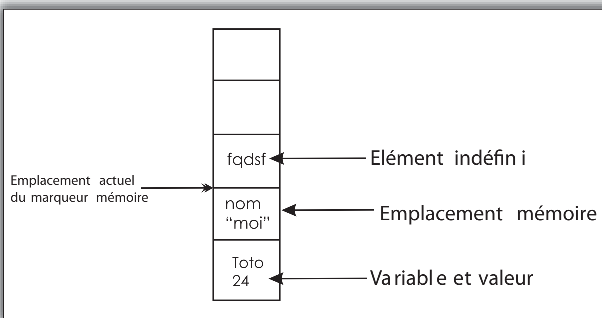


Figure 9.23 : *Pile mémoire avec le marqueur*

Quand vous déclarez une variable, un emplacement mémoire va être utilisé, et ce marqueur va être déplacé :

```
Dim X As Integer
```

Le programme pourra alors accéder aux parties de la mémoire qui sont au-dessus de ce marqueur. `X` vient d'être déclarée, elle est au-dessus du marqueur, on peut l'utiliser. Cependant, vous n'avez pas donné à `X` de valeur initiale. Entre sa déclaration et l'entrée dans la boucle, on ne sait pas ce que vaut `X`. Lorsque l'on déclare des variables, le marqueur se déplace vers le bas, donnant ainsi l'accès à de nouvelles variables.

Or, ce marqueur se déplace aussi vers le haut, bloquant ainsi l'accès à d'anciennes variables. Il se déplace essentiellement vers le haut lorsque l'on sort d'un bloc. En Visual Basic, on peut dire que la fin d'un bloc correspond à un endroit où il y a un `End`.

Examinons le programme précédent. On déclare X tout d'abord. Un accès lui est donné dans la pile et le marqueur descend donc.

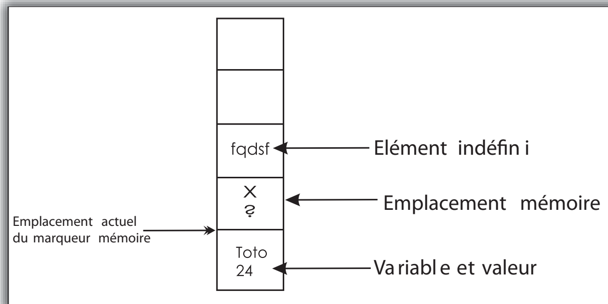


Figure 9.24 : *Pile mémoire, avec X accessible*

On entre dans un nouveau bloc, la boucle For. La position du marqueur est enregistrée pour que l'on puisse y revenir quand on sortira de ce bloc.

```
Dim Y As Integer
```

Ensuite, on déclare Y. Encore une fois, le marqueur descend et un emplacement mémoire est créé pour Y :

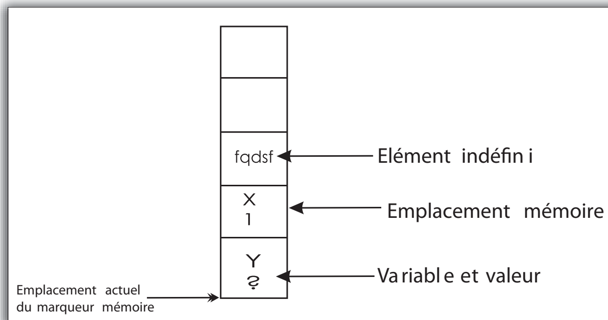


Figure 9.25 : *Pile mémoire, avec X et Y accessibles*

Les traitements sont faits avec cet environnement. L'environnement est l'ensemble des points accessibles à partir de là où l'on est. Ces points peuvent être des fonctions, des méthodes, des variables, etc.

Enfin, on quitte la boucle For. On quitte donc un bloc et le marqueur remonte à l'endroit enregistré lorsque l'on est entré dans cette boucle.

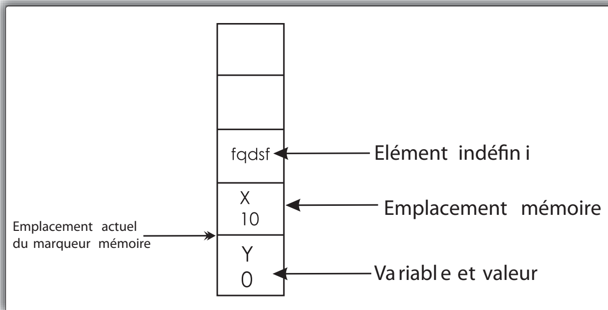


Figure 9.26 : *Pile mémoire, après retour dans l'ancien bloc*

L'emplacement mémoire correspondant à `Y` étant maintenant en dessous du marqueur, il n'est plus dans l'environnement courant et n'est donc plus accessible. Bien que `Y` ne soit pas accessible, son emplacement mémoire existe encore, avec sa valeur, tant que l'état de la pile n'a pas changé.

L'environnement change souvent : à l'entrée ou à la sortie d'une fonction, à l'entrée et à la sortie d'une structure de contrôle. La mémoire est en constants mouvements.

Désormais, vous devriez mieux comprendre les concepts de portée et d'accessibilité. C'est un sujet vaste et intéressant. Nous vous invitons à l'approfondir pour découvrir justement les subtilités que nous n'exposons pas ici.

Les constructeurs d'objets

Lorsque nous avons présenté les mécanismes de la programmation orientée objet, il a été question du mot-clé `New` dans le cadre de la création d'un objet. Ce mot-clé permet d'instancier un objet.

Qu'est-ce que l'instanciation ? C'est le mécanisme qui permet de créer un objet, une instance de classe. Son rôle réel est en fait de réserver un espace mémoire pour l'utilisation de l'objet et, le cas échéant, d'initialiser certains de ses attributs. Cependant, la gestion de la mémoire n'est pas la même pour les objets et pour les variables locales. En effet, un objet doit rester en mémoire jusqu'à ce que vous n'en ayez plus besoin. Il ne peut alors pas être stocké dans la pile. Les objets sont stockés dans le tas managé.

Le tas managé est un gros espace de mémoire réservé au programme. Celui–ci pourra y stocker tous les objets nécessaires à son exécution. Lorsqu'une classe est instanciée, le programme lui crée un espace mémoire dans le tas managé. Cela se fait grâce au constructeur.

```
Dim monChasseur As New AvionDeChasse()
```

Le constructeur est la partie qui suit `New`. C'est une méthode qui a pour nom celui de la classe et lui seul permet d'instancier cette classe. Il existe un constructeur par défaut. C'est pourquoi vous n'avez pas eu à l'indiquer lors de l'écriture de la classe `AvionDeChasse`. Dans ce cas, il réserve juste l'espace mémoire. Cependant, il est possible de redéfinir le constructeur, et ainsi de pouvoir faire un certain nombre d'opérations lors de l'instanciation. Vous pouvez modifier sa signature, en ajoutant des paramètres par exemple. Il se crée comme une méthode qui ne renvoie aucune valeur de retour et dont le nom est `New`. Appliquons cela à l'avion de chasse :

```
Public Class AvionDeChasse
' Nous ne gardons qu'une partie de la classe
' pour cette illustration
  Private vitesseMax As Integer
  Private nbreAilerons As Integer
Private reserveKerozene As Integer

  Public Sub afficheKero()
    MessageBox.Show("Niveau kéro : " & Me.reserveKerozene)
  End Sub

  Public Sub New(ByVal keroOriginal As Integer)
Me.reserveKerozene = keroOriginal
End Sub
End Class
```

On redéfinit le constructeur en lui ajoutant un paramètre qui permettra d'initialiser son niveau de kérosène à l'instanciation. Pour appeler ce constructeur, il suffira de tenir compte de la nouvelle signature, et donc d'ajouter ce paramètre :

```
Dim monChasseur As New AvionDeChasse(100)
monChasseur.afficheKero()
```

Un objet `AvionDeChasse` a été créé avec un niveau de kérosène valant 100. Cela est confirmé par la boîte de dialogue, qui affichera "*Niveau kéro : 100*".

Récupération de la mémoire : le ramasse-miettes

Comme nous l'avons expliqué, le tas managé est un espace de mémoire réservé à l'application, dans lequel le programme va pouvoir stocker les objets qu'il utilise. Cependant, cet espace est limité, et vous n'avez pas donné de limite au nombre possible d'objets à instancier.

Pourtant il y en a une : celle de la capacité du tas managé. Si celui-ci est trop plein et que vous vouliez instancier des objets, une exception `OutOfMemoryException` est levée. On a un espace mémoire, on peut réserver des emplacements dans cet espace, mais il faut pouvoir libérer ceux dont on ne se sert pas.

Cette tache est effectuée par un mécanisme appelé "*ramasse-miettes*" (*garbage collector*), qui s'occupe de libérer l'espace mémoire occupé par des objets dont vous ne vous servez plus. Le programme sait quels sont les objets que vous allez pouvoir utiliser ou pas. Si vous pouvez les utiliser, on dit qu'ils sont référencés. Dans le cas contraire, on dit qu'ils sont déréférencés.

Prenons ce morceau de programme relatif aux avions :

```
Public Sub essaiDereferencement()
  Dim monChasseur2 As New AvionDeChasse()
End Sub

Dim monChasseur1 As New AvionDeChasse()
essaiDereferencement()
```

On déclare une fonction qui instancie un chasseur, puis on instancie un autre chasseur et l'on fait appel à cette fonction.

À la fin du programme, en admettant qu'il continue après, on pourra toujours utiliser `monChasseur1`. `monChasseur1` est donc référencé.

Par contre `monChasseur2` est instancié dans la fonction `essaiDereferencement`. Or, en sortant de cette fonction, on ne peut plus y accéder. En effet, il a été déclaré dans une fonction, et bien qu'il existe en mémoire dans le tas managé, sa portée ne peut aller au-delà de la fonction `essaiDereferencement`. À la fin de ce bout de programme, vous ne pouvez plus atteindre `monChasseur2`, il est déréfencé.

Le ramasse-miettes vérifie les objets qu'il a en charge et détermine ceux qui sont référencés et ceux qui ne le sont plus. Étant donné que ceux qui sont déréférencés ne sont plus accessibles, garder leur espace mémoire est inutile. Celui-ci est alors libéré pour permettre la création de nouveaux objets.

Les destructeurs d'objets

Le ramasse-miettes est un mécanisme qui permet de gérer automatiquement la mémoire. Il permet d'éviter les fuites de mémoire et enlève au programmeur une grosse part de réflexion. En effet, dans les langages de programmation où il faut gérer la mémoire manuellement, il n'est pas rare que l'espace dédié à certains objets ne soit pas libéré, provoquant ainsi des fuites de mémoire. Le programmeur doit savoir où sont utilisés ses objets, où ils ne le sont plus, et à ce moment-là libérer la mémoire. Grâce au ramasse-miettes, cela peut être évité.

Mais, il arrive que l'on ait besoin de libérer manuellement la mémoire, ou encore que l'on veuille effectuer une opération particulière juste avant que celle-ci ne soit libérée. Si l'on reprend l'analogie avec l'avion, la libération de la mémoire serait la destruction de celui-ci, mais on aimerait pouvoir lancer le siège éjectable juste avant, et ainsi épargner la vie du pilote !

Imaginons que vous vouliez effectuer une opération, juste avant la destruction de l'objet. On entend par destruction de l'objet sa libération. Il faut savoir que tous les objets possèdent une méthode `Finalize`. En effet, tous les objets en Visual Basic .NET dérivent implicitement de la classe `Object` ; celle-ci définit une méthode `Finalize`, qui est `Overridable`. Cette méthode est appelée lorsque le ramasse-miettes libère un objet. En redéfinissant cette méthode (avec le mot-clé `Overrides`, en tant que membre protégé), vous pouvez effectuer les instructions que vous voulez lorsque l'objet est détruit :

```
Public Class AvionDeChasse
  Public Sub siegeEjectable()
  ' L'implémentation n'est pas nécessaire
  ' pour la compréhension
  End Sub

  Protected Overrides Sub Finalize()
    Me.siegeEjectable()
  End Sub
End Class
```

`Finalize` étant redéfinie, la méthode `siegeEjectable` sera appelée lorsque l'objet sera détruit. Attention cependant, il y a quelques règles à respecter. Il ne faut pas appeler explicitement `Finalize`, ce sera fait par le ramasse-miettes. Si l'on veut effectuer la libération de l'objet manuellement, il y a une manière valable d'opérer, décrite ci-après. De plus, il ne faut pas qu'une exception soit levée. Si c'est l'objet qui gère l'exception, elle ne pourra pas être récupérée, car celui-ci sera détruit ; cela peut provoquer l'arrêt du programme.

De plus, le ramasse-miettes libère la mémoire selon son bon vouloir. En redéfinissant la méthode, vous ne pouvez pas forcer son utilisation, qui se fera au moment où le ramasse-miettes l'aura décidé, par exemple quand un besoin de mémoire se fera sentir, ou alors si le nombre d'objets déréférencés est trop important.

Si vous voulez forcer la libération, il faut passer par l'implémentation d'un modèle de conception, le `DisposeFinalize`. Nous présenterons les modèles de conception plus loin dans ce chapitre, car nous verrons juste après comment gérer la sauvegarde et le chargement de vos objets de manière efficace. Nous vous invitons donc à laisser faire le ramasse-miettes dans un premier temps, car quoique l'on puisse en dire, celui-ci est efficace dans la gestion de la mémoire.

9.3. Enregistrer automatiquement vos objets : la sérialisation

Gérer les données peut devenir très fastidieux quand votre programme commence à devenir un peu important. Imaginez si vous deviez opter pour un enregistrement et un chargement à partir d'un fichier textuel... Pour chacun des attributs que vous avez à gérer, il vous faut créer une méthode de sauvegarde ou de chargement ou alors adapter celle qui est existe déjà. Cela limite un peu la croissance de vos structures ou alors au prix de quels efforts...

On a également vu qu'il est possible de les enregistrer dans des bases de données. Là encore, le travail peut se révéler important : installer le serveur, le configurer, s'assurer de son bon fonctionnement... Ceci n'est pas très efficace pour des applications indépendantes.

En revanche, nous avons également vu les notions d'objet et de classe, pour lesquelles le framework .Net définit et implémente un mécanisme

de sauvegarde et de chargement automatique. Vous pourrez en user et en abuser, à tel point que cela deviendra vite le moyen privilégié pour la gestion de la sauvegarde de vos objets.

Dans un premier temps, nous allons voir plus en détail ce qu'est la sérialisation et ses différents types. Nous préparerons également une petite application de test. Puis nous étudierons de manière plus détaillée deux types de sérialisations : binaires et XML.

Qu'est-ce que la sérialisation ?

Fort de vos connaissances concernant les enregistrements sur les bases de données et les fichiers, vous savez gérer des données, pas de souci...



*Reportez-vous à ce sujet au chapitre **Enregistrer des données**.*

Mais on admettra volontiers que, si on pouvait s'affranchir de tout ce travail pour pouvoir utiliser son temps à des parties de programmes plus intéressantes et plus constructives, ce serait quand même beaucoup plus sympa. La sérialisation nous permet cela. Pourquoi ? Car en entrée elle prend un objet, et elle vous ressort un fichier. La désérialisation, son opération opposée, prend ce fichier et vous renvoie votre objet. Qu'avez-vous fait pour cela ? Rien, si ce n'est avoir écrit les méthodes de sérialisation et de désérialisation. Et elles ne bougeront plus même si votre classe bouge avec des milliers d'attributs et de propriétés en plus.

Il existe plusieurs types de sérialisations :

- **Binaire**. Enregistre vos objets dans un format machine non lisible par l'homme. C'est la première que nous allons voir.
- **XML**. Enregistre vos objets au format XML, standard, et lisible par l'homme. Nous l'étudierons également.
- **SOAP**. On peut grossièrement dire qu'il s'agit d'une version améliorée de la version XML. C'est en fait un peu plus que cela, mais nous ne l'étudierons pas dans cet ouvrage.

Afin de vous présenter cette technique, nous allons créer une petite application qui se fondera sur une classe que nous allons sérialiser et désérialiser, la classe `Livre` :

```
Public Class Livre
    Private mTitre As String
    Private mAuteur As String
    Private mEditeur As String
    Private mAnneeDeParution As Integer
    Private mNbreDePages As Integer
    Private mIsbn As Long

    Property Titre() As String
        Get
            Return mTitre
        End Get
        Set(ByVal Value As String)
            mTitre = Value
        End Set
    End Property

    Property Auteur() As String
        Get
            Return mAuteur
        End Get
        Set(ByVal Value As String)
            mAuteur = Value
        End Set
    End Property

    Property Editeur() As String
        Get
            Return mEditeur
        End Get
        Set(ByVal Value As String)
            mEditeur = Value
        End Set
    End Property

    Property AnneeDeParution() As Integer
        Get
            Return mAnneeDeParution
        End Get
        Set(ByVal Value As Integer)
            mAnneeDeParution = Value
        End Set
    End Property

    Property NombreDePage() As Integer
        Get
            Return mNbreDePages
        End Get
        Set(ByVal Value As Integer)
            mNbreDePages = Value
        End Set
```

```
    End Property

    Property ISBN() As Long
        Get
            Return mIsbn
        End Get
        Set(ByVal Value As Long)
            mIsbn = Value
        End Set
    End Property
End Class
```

Cette classe définit six attributs membres (titre, auteur, année de parution, ISBN, nombre de pages, éditeur) ainsi que les propriétés (en lecture / écriture) associées.

Entrons maintenant dans le vif du sujet. Nous allons créer une légère application pour manier ces mécanismes.

1 Tout d'abord, on crée un nouveau projet.



Figure 9.27 : *Création du nouveau projet*

2 C'est un projet `WindowsForms`, et `ProjetSerialisation` semble être un nom adéquat (voir Figure 9.28).

3 Il faut ajouter maintenant au projet la classe `Livre`, qui sera le point central du projet (voir Figure 9.29, 9.30).

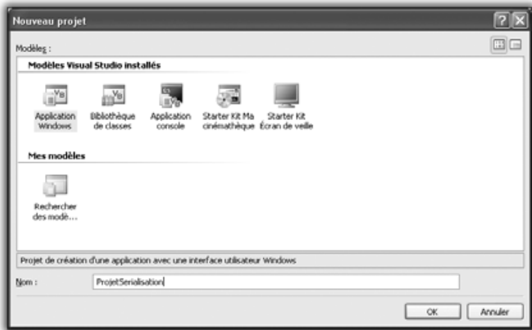


Figure 9.28 : *Choix du nom et du type de projet*

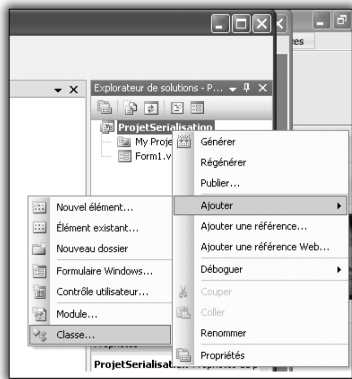


Figure 9.29 : *Ajout d'une nouvelle classe*

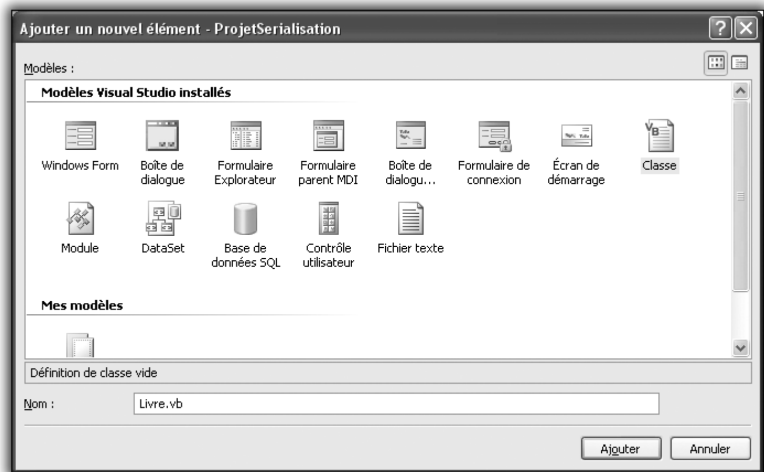


Figure 9.30 : *On la nomme Livre*

4 Créez maintenant une interface graphique, à base de `Label`, de `TextBox` et de `Bouton`, afin qu'elle ressemble à quelque chose dans ce style :



Figure 9.31 :
Interface du projet



Reportez-vous à ce sujet au chapitre ***Dialoguer avec un ordinateur****.*

Maintenant que la base du programme est faite, nous allons pouvoir attaquer en profondeur et voir en détail comment manier sérialisation et désérialisation.

Les différentes méthodes de sérialisation

Nous allons étudier dans cette partie seulement deux des sérialisations existantes :

- la sérialisation binaire ;
- la sérialisation XML.

La sérialisation binaire

La sérialisation binaire est un type de sérialisation qui transforme votre objet en fichier binaire, c'est-à-dire directement en code machine. Globalement, si vous essayez de lire, vous n'y arriverez pas. En tout cas pas en entier.



Figure 9.32 :
Fichier binaire ouvert avec un éditeur de texte

Comme vous pouvez le constater, on ne comprend pas grand-chose, mais il est possible de discerner certaines chaînes de caractères. En effet, que ce soit dans un programme ou dans un fichier, le code machine est le même... Mais ce n'est pas vraiment l'objet de ce chapitre.

Tout d'abord, il faut savoir que, pour sérialiser et désérialiser, il faut un sérialiseur, qui fait aussi la désérialisation. Logique ! Le sérialiseur binaire est fourni par le framework .Net, dans le namespace `Binary`. Il vous faudra donc inclure ce namespace dans votre code.

```
Imports System.Runtime.Serialization.Formatters.Binary
```

Enregistrement : sérialisation

Une méthode de sérialisation écrit un fichier à partir d'un objet. Elle prendra donc cet objet en paramètre, ici un `Livre`, et ne renverra rien. Mais nous allons quand même lui faire renvoyer un booléen pour nous dire si l'opération s'est bien passée. Elle aura donc une signature comme ceci :

```
Private Function BinarySerialize(ByVal monLivre As
✂ Livre) As Boolean
End Function
```

Comme nous allons écrire dans un fichier, il faut créer celui-ci, par exemple `Livre.bin`. Il nous faut donc également ajouter le namespace `IO`.



Reportez-vous à ce sujet au chapitre ***Enregistrer des données, les fichiers****.*

Le sérialiseur binaire est un `BinaryFormatter` qui possède un constructeur par défaut. Nous pouvons donc enrichir notre méthode de sérialisation avec ces nouveaux éléments :

```
Private Function BinarySerialize(ByVal monLivre As
✂ Livre) As Boolean
    If monLivre Is Nothing Then Return False
    Dim stream As FileStream = New FileStream("Livre
   ✂ .bin", FileMode.Create)
    Return True
End Function
```



Validité de l'objet

Attention à ne pas oublier la vérification de la validité de l'objet (`If monLivre Is Nothing Then Return False`), sans quoi vous risquez de provoquer une exception de type `NullReferenceException`.

La dernière étape de cette méthode est la sérialisation effective. Elle est réalisée par le sérialiseur et prendra comme paramètre le fichier dans lequel écrire, ainsi que l'objet à sérialiser :

```
serializer.Serialize(stream, monLivre)
```

Il ne faudra pas oublier de libérer les ressources utilisées pour le fichier, par l'appel à la méthode `Close`. Au final, la méthode de sérialisation ressemblera à ceci :

```
Private Function BinarySerialize(ByVal monLivre As
✂ Livre) As Boolean
    If monLivre Is Nothing Then Return False
    Dim stream As FileStream = _
             New FileStream("Livre.bin", FileMode
            ✂ .Create)
    Dim serializer As New BinaryFormatter
    serializer.Serialize(stream, monLivre)
    stream.Close()
    Return True
End Function
```

Liez maintenant cette méthode au bouton de sauvegarde de votre interface. Et en avant pour le test !

Démarrez le projet et remplissez les différents champs avec les valeurs de votre choix.

Figure 9.33 : *Description d'un livre*

Cliquez sur sauvegarder, et là, surprise !

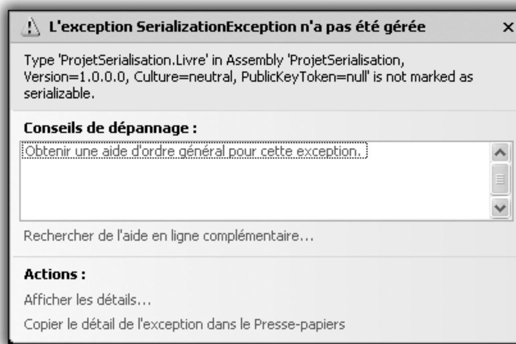


Figure 9.34 : *Erreur de sérialisation*

Erreur de sérialisation ! Votre programme ne peut plus continuer à s'exécuter, car l'objet que l'on veut sérialiser est l'instance d'une classe qui n'est pas marquée comme sérialisable.

Qu'à cela ne tienne ! Marquons-la sérialisable en y ajoutant un attribut.

```
<Serializable()> _
Public Class Livre
    Private mTitre As String
    Private mAuteur As String
    Private mEditeur As String
    Private mAnneeDeParution As Integer
    Private mNbreDePages As Integer
    Private mIsbn As Long

    ReadOnly Property Titre() As String
        Get
            Return mTitre
        End Get
        'Set(ByVal Value As String)
        '    mTitre = Value
        'End Set
    End Property

    ReadOnly Property Auteur() As String
        Get
            Return mAuteur
        End Get
        'Set(ByVal Value As String)
        '    mAuteur = Value
        'End Set
    End Property
```

```
    ReadOnly Property Editeur() As String
        Get
            Return mEditeur
        End Get
        'Set(ByVal Value As String)
        '    mEditeur = Value
        'End Set
    End Property

    Property AnneeDeParution() As Integer
        Get
            Return mAnneeDeParution
        End Get
        Set(ByVal Value As Integer)
            mAnneeDeParution = Value
        End Set
    End Property

    Property NombreDePage() As Integer
        Get
            Return mNbreDePages
        End Get
        Set(ByVal Value As Integer)
            mNbreDePages = Value
        End Set
    End Property

    Property ISBN() As Long
        Get
            Return mIsbn
        End Get
        Set(ByVal Value As Long)
            mIsbn = Value
        End Set
    End Property
End Class
```

Vous pouvez maintenant réessayer. Et vous obtiendrez en sortie un fichier *Livre.bin* qui contiendra ceci :

Figure 9.35 : *Fichier de sortie binaire*

Nous remarquons que certaines chaînes de caractères sont lisibles. Certaines ont été générées pour les besoins du framework .Net (nom du projet, version...) et d'autres sont les valeurs des attributs de votre objet. On peut y voir le titre, l'auteur, l'éditeur, mais remarquez qu'on ne voit pas les chiffres. Les types numériques en code machine ne sont pas lisibles, contrairement aux chaînes de caractères. Ils correspondent à certains des petits carrés que vous voyez.

Chargement : désérialisation

L'opération contraire à la sérialisation, qui permet la création de votre objet préalablement enregistré *via* la sérialisation, est la désérialisation. En toute logique, vous pensez qu'elle se fait avec un désérialiseur. Et vous avez raison de penser cela, sauf qu'elle se fait également en utilisant le sérialiseur.

En revanche, une méthode de désérialisation a une signature totalement différente. En effet, nous avons vu que la désérialisation consistait à utiliser un fichier pour créer un objet, ici de type `Livre`. Et, comme nous savons quel fichier aller chercher, cette méthode ne nécessite pas de paramètres. Voici donc la base de notre méthode de désérialisation :

```
Imports System.Runtime.Serialization.Formatters.Binary

    Private Function BinaryDeserialize() As Livre
        Dim monLivre As Livre = Nothing

        Dim deserializer As New BinaryFormatter

        Return result
    End Function
```

De la même manière que pour enregistrer, il va falloir ici gérer un fichier, sauf qu'on va l'ouvrir et non le créer.



Fichier manquant

Attention à bien vérifier l'existence du fichier, sans quoi une exception de type `IOException` peut arrêter votre programme !

```
    Private Function BinaryDeserialize() As Livre
        Dim fichier As String = "Livre.bin"
        Dim monLivre As Livre = Nothing

        If Not File.Exists(fichier) Then Return Nothing
        Dim deserializer As New BinaryFormatter
```

```
        Return result
    End Function
```

Il faut maintenant désérialiser et c'est encore plus facile que la sérialisation. Le seul paramètre nécessaire est le fichier à lire. La méthode qui fait cela est `Deserialize`. Là encore, il ne faudra pas oublier de libérer les ressources du fichier, sinon vous ne pourrez plus l'ouvrir avec un autre programme tant que notre programme de test sera actif. La méthode finale ressemble à ceci :

```
    Private Function BinaryDeserialize() As Livre
        Dim fichier As String = "Livre.bin"
        Dim monLivre As Livre = Nothing

        If Not File.Exists(fichier) Then Return Nothing

        Dim deserializer As BinaryFormatter
        Dim stream As FileStream = New FileStream(fichier,
      ✂ FileMode.Open)
        monLivre = deserializer.Deserialize(stream)
        stream.Close()

        Return monLivre
    End Function
```

Maintenant, il vous suffit de charger sur votre interface graphique les informations récupérées à partir de l'instance de `Livre` chargée pour vérifier le chargement :



Figure 9.36 : *Objet chargé*

On retrouve bien notre objet préalablement enregistré. Quelque part, c'est rassurant, n'est-ce pas ?

Comme vous le voyez, il est très facile de charger et d'enregistrer des instances de classe grâce à la sérialisation. N'hésitez surtout pas à

utiliser ces méthodes. Elles vous feront gagner un temps précieux sur la gestion de vos données, temps que vous pourrez utiliser pour des choses de plus haut niveau.

Voyons maintenant la sérialisation XML, qui, à l'utilisation, sera quasiment identique à notre sérialisation binaire, mais qui aura un fichier de sortie totalement différent.

La sérialisation XML

Je ne reviendrai pas ici sur ce qu'est le XML mais, pour ceux qui auraient pris le livre en cours, il faut savoir que c'est un langage à balises permettant de décrire des éléments suivant une hiérarchie donnée.



Reportez-vous à ce sujet au chapitre ***Enregistrer des données, Sauvegarde et fichier : le XML.***

Mais, comme rien ne vaut un exemple, voici un extrait de fichier XML :

```
<?xml version="1.0" encoding="UTF-8"?>
  <!-- '''Commentaire''' -->
  <élément-document xmlns="http://exemple.org/" xml:lang=";fr">
    <élément>Texte</élément>
    <élément>élément répété</élément>
    <élément>
      <élément>Hiérarchie récursive</élément>
    </élément>
    <élément>Texte avec<élément>élément</élément>mêlé</élément>
    <élément/><!-- élément vide -->
    <élément attribut="valeur"></élément>
  </élément-document>
```

Figure 9.37 : *Exemple de fichier XML*

Vous remarquez que, contrairement au format binaire, le format XML est totalement lisible. *A fortiori*, il permet de savoir ce que l'on fait. Nous verrons que cela pourra devenir un moyen de modifier nos données à la volée.

De la même manière que pour la sérialisation et la désérialisation binaire, il faut un sérialiseur. Dans ce cas-là, il est dans le namespace `Xml.Serialization`. Il faut donc ajouter ce namespace au projet.

```
Imports System.Xml.Serialization
```

Enregistrement : sérialisation

Le principe de sérialisation, qu'il soit binaire, XML ou même SOAP (que nous ne traiterons pas ici) reste toujours le même : on crée un fichier à partir d'un objet. C'est pourquoi la signature de notre méthode de sérialisation ne va pas changer. Elle prendra toujours en paramètre un objet `Livre` et renverra toujours un booléen qui nous dira si l'opération s'est bien passée.

```
Private Function XMLSerialize(ByVal monLivre As Livre)
✂ As Boolean
End Function
```

Jusque-là tout est identique, si ce n'est le nom de la méthode. (Ceci dit, on aurait pu garder le même nom, ce qui aurait été très maladroit, car le nom ne correspondrait plus à l'action.)

Une fois encore, nous allons écrire dans un fichier, exactement de la même manière que pour la sérialisation binaire, mais on va lui donner un autre nom. `.xml` étant le standard pour définir un fichier XML, nous l'appelons *Livre.xml*.



Besoin de ressources

Nous n'oublierons pas de libérer les ressources du fichier ni de vérifier la validité de l'objet que nous voulons sérialiser.

```
Private Function XMLSerialize(ByVal monLivre As Livre)
✂ As Boolean
    If monLivre Is Nothing Then Return False
    Dim stream As FileStream = _
      New FileStream("Livre.xml", FileMode.Create)
    stream.Close()
    Return True
End Function
```

Il nous faut maintenant instancier un sérialiseur. Cependant, dans le cas du sérialiseur XML, il faut lui passer comme paramètre le type de l'objet que nous voulons sérialiser, tout simplement en utilisant `GetType`. L'utilisation se fait ensuite de manière totalement identique à un sérialiseur binaire, avec comme paramètre le lien vers le fichier et l'objet à sérialiser :

```
Private Function XMLSerialize(ByVal monLivre As Livre)
✂ As Boolean
    If monLivre Is Nothing Then Return False
    Dim stream As FileStream =
```

```
                New FileStream("Livre.xml", FileMode
               ✂ .Create)
        Dim serializer As XmlSerializer = _
          New XmlSerializer(GetType(Livre))
        serializer.Serialize(stream, monLivre)
        stream.Close()
        Return True
    End Function
```

Nous obtiendrons donc un fichier de sortie, au format XML, contenant les informations de notre objet, qui, lui, est totalement lisible.

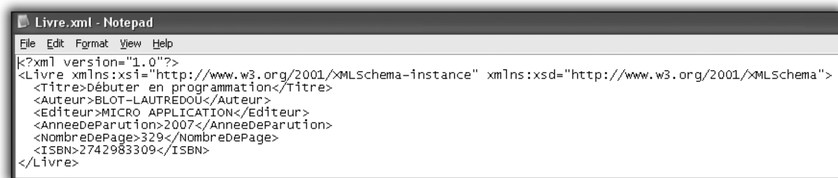


Figure 9.38 : *Fichier XML de notre objet sérialisé*

Chargement : désérialisation

Cette fois encore, la désérialisation sera avec un objet sérialiseur. Cela a au moins cet avantage qu'on l'instancie de la même manière.

De plus, nous gérerons également un fichier à ouvrir, que l'on prendra bien soin de refermer pour éviter de laisser un lien qui empêchera son utilisation par un autre programme. On n'oubliera pas non plus de vérifier son existence pour éviter des erreurs `IOException`. Voici la méthode de désérialisation XML :

```
    Private Function XMLDeserialize() As Livre
        Dim fichier As String = "Livre.xml"
        Dim monLivre As Livre = Nothing

        If Not File.Exists(fichier) Then Return Nothing

        Dim deserializer As XmlSerializer = _
        New XmlSerializer(GetType(Livre))
        Dim stream As FileStream = New FileStream(fichier,
       ✂ FileMode.Open)
        Dim result As Object = Nothing
        monLivre = deserializer.Deserialize(stream)
        stream.Close()

        Return monLivre
    End Function
```

Comme la méthode de désérialisation est quasiment identique en binaire ou en XML, je ne vais pas détailler une nouvelle fois les différents points de la méthode. Cependant, la sérialisation XML a certaines particularités.

Si vous chargez directement le fichier préalablement enregistré, vous obtiendrez le même objet que celui d'origine. Jusque-là, normal, nous pouvons vérifier dans votre interface de chargement :

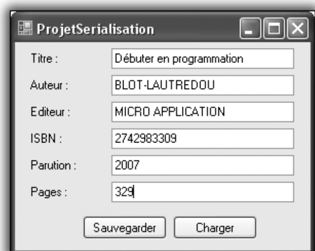


Figure 9.39 :
Objet chargé

Maintenant, étant donné que le fichier XML est lisible et modifiable, essayez de changer certaines de ses valeurs :



Figure 9.40 :
Modification du fichier XML à la main

Et, là, si vous tentez de recharger le même fichier, voici ce que vous obtiendrez dans votre interface :



Figure 9.41 :
Interface obtenue à partir du fichier XML modifié

Comme vous le voyez, les éléments ont changé, prenant maintenant les valeurs contenues dans le fichier XML. Quelque part, c'est rassurant, car cela nous prouve que la désérialisation fonctionne. D'un autre côté, cela ouvre énormément de possibilités pour piloter des objets (et donc une application) à partir de fichiers. Par exemple, créons un objet `MailSender`, dont le but est d'envoyer des courriers électroniques. Ajoutons-lui des attributs "*Nombre de message*", "*destinataire*", "*corps*"... En le pilotant par sérialisation et désérialisation, il n'y a même plus besoin de modifier notre programme pour faire changer son comportement. On le définit dans le fichier XML de sérialisation et il sera à même de trouver les bonnes valeurs automatiquement. Je vous laisse le soin de créer un tel projet, cela pourrait être un très bon exercice de mise en pratique, pas trop difficile, concret et intéressant.

Modifions quelque peu notre code. Pour des raisons de sécurité, nous souhaiterions que certaines des propriétés de notre classe `Livre` soient en lecture seule (par exemple le titre, l'auteur et l'éditeur... Eh oui, on ne peut pas nous changer) :

```
ReadOnly Property Titre() As String
    Get
        Return mTitre
    End Get
End Property

ReadOnly Property Auteur() As String
    Get
        Return mAuteur
    End Get
End Property

ReadOnly Property Editeur() As String
    Get
        Return mEditeur
    End Get
End Property
```

Maintenant, retentez un chargement et vous obtiendrez ceci : (voir Figure 9.42)

Les champs dont les propriétés étaient en lecture seule n'apparaissent plus... En effet, c'est une particularité de la sérialisation XML. Elle se fonde sur les propriétés et utilise les `Get` et les `Set` pour sérialiser ou désérialiser le cas échéant. Alors que la sérialisation binaire sérialisera et désérialisera les attributs directement, quelle que soit leur accessibilité.

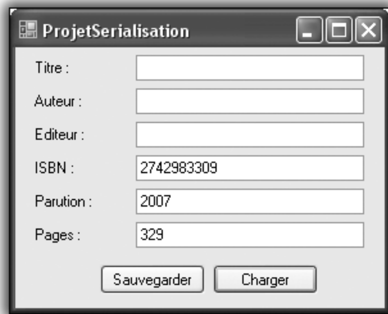


Figure 9.42 :
Résultat avec des propriétés en lecture seule

Quoi qu'il en soit, qu'elle soit binaire ou XML, n'hésitez pas à user et à abuser de la sérialisation. Choisissez celle qui vous convient selon le besoin de votre application. Si vous devez modifier des données à la volée, préférez la sérialisation XML car elle est lisible par l'homme. En revanche, étant donné que c'est du texte, elle sera moins performante et prendra plus de place en mémoire. D'un autre côté, la sérialisation binaire est plus performante, et elle ne nécessite pas de définir des propriétés. Elle est plus adaptée pour une utilisation directe qui ne demande pas de modification entre la sauvegarde et le chargement. Vous verrez, une fois acquise, la sérialisation est un concept dont on ne peut plus du tout se passer tellement elle facilite la gestion de la sauvegarde et du chargement des données.

9.4. Les modèles de conception

Un modèle de conception permet de résoudre un problème récurrent en tirant parti des mécanismes de la programmation orientée objet. Certains peuvent être simples, comme le `Singleton` que nous détaillerons et implémenterons complètement, mais ils sont le plus souvent assez subtils. Par contre, il garantisse une solution au problème, et ce de manière toujours propre. C'est pourquoi, il est fortement recommandé de les utiliser lorsque vous rencontrez un problème pour lequel il existe un modèle de conception.

Dans cette partie, vous verrez en détail le modèle de conception `Singleton`, qui permet de garantir l'unicité d'une instance de classe dans tout le programme et vous l'implémenterez. Puis, vous étudierez quelques-uns des modèles de conception les plus courants, ce qui vous donnera un point de départ si vous voulez approfondir le sujet.

Implémentation du modèle Singleton

Le `Singleton` est le modèle le plus souvent utilisé pour présenter le principe des modèles de conception. C'est l'un des plus faciles à mettre en œuvre, c'est pourquoi il est compréhensible et utilisable même pour des débutants.

Il permet de garantir l'unicité de l'instance d'une classe et son accessibilité à partir de n'importe endroit du programme. Les cas de figure où une instance unique présente un intérêt sont multiples. Par exemple, dans la programmation d'un jeu, le moteur de son, chargé de gérer la lecture des sons et des musiques, doit être unique. Vu son utilisation omniprésente, il doit être accessible de partout.

Ce modèle fait appel à plusieurs principes de la programmation orientée objet. L'idée principale du `Singleton` est que l'instance est englobée dans la classe. Elle sera donc un attribut de cette classe. Cet attribut devra être privé, ce qui empêchera son accès à tout élément extérieur à la classe. De plus, il devra être initialisé à `Nothing` pour que l'on puisse vérifier son existence. Pour pouvoir y accéder, on devra implémenter un accesseur. Cet accesseur devra être partagé. De cette manière, on pourra y accéder de partout, en invoquant la classe. De plus, le constructeur devra être privé. De cette manière, on empêchera une instanciation manuelle de l'objet. Or, si le constructeur est privé, comment créer l'instance de la classe ? Cela sera fait dans l'accesseur. Celui-ci devra vérifier si l'instance de la classe existe. Si elle existe, il la retournera. Dans le cas contraire, il l'instanciera et la retournera. Cet accesseur est la clé du modèle. En effet, c'est lui qui vérifie l'existence et l'unicité de l'instance, et la renvoie le cas échéant. Voici un schéma de ce modèle.

La première étape consiste à déclarer la classe avec son constructeur privé, pour empêcher l'instanciation :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub
End Class
```

Le constructeur peut contenir d'autres opérations à effectuer, par rapport à l'utilisation de la classe.

On crée ensuite l'instance de la classe. La particularité du `Singleton` est que l'instance est un attribut privé de cette classe. Celui-ci doit être

également partagé, car l'accesseur sera également partagé pour être accessible à partir de la classe, et non d'un objet. Il doit être initialisé à `Nothing`. De cette manière, on saura s'il existe ou pas. S'il vaut `Nothing`, cela signifie qu'il n'existe pas. Sinon il existe :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing
End Class
```

Il faut ensuite implémenter l'accesseur qui permettra de récupérer cette instance. Si l'instance n'existe pas, c'est-à-dire si l'attribut `instance` vaut `Nothing`, cet accesseur doit créer l'instance. Il le pourra car il a accès au constructeur, même si celui-ci est privé. Ensuite, il retournera l'instance.

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing

' L'accesseur crée l'instance si elle n'existe pas
' Puis il la retourne
Public Shared Function getInstance() As Singleton
  If (Me.instance Is Nothing)
    Me.instance = New Singleton()
  End If
  Return Me.instance
End Function

End Class
```

Le `Singleton` est maintenant totalement défini. Pour pouvoir y avoir accès et l'utiliser, il suffit d'invoquer la classe et de faire appel à sa méthode `getInstance` :

```
Singleton.getInstance()
```

Bien qu'il existe, le `Singleton` ne fait rien pour l'instant. On lui donne un nom, que l'on va modifier et réafficher, de manière à vérifier l'unicité de l'instance :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing

' L'accesseur crée l'instance si elle n'existe pas
' Puis il la retourne
Public Shared Function getInstance() As Singleton
  If (Me.instance Is Nothing)
    Me.instance = New Singleton()
  End If
  Return Me.instance
End Function

  Public nom As String = "Toto"
End Class
```

Le `Singleton` s'appelle `Toto` à l'origine. On modifie son nom que l'on réaffiche pour savoir si cela a affecté la bonne personne :

```
MessageBox.Show("nom du singleton : " & _
 Singleton.getInstance().nom)
Singleton.getInstance().nom = "Bibi"
MessageBox.Show("nom du singleton : " & _
 Singleton.getInstance().nom)
```

La première boîte de dialogue affiche le nom d'origine, "*Toto*". Ensuite, on modifie celui-ci. D'après le principe du `Singleton`, il n'y a qu'une instance. Par conséquent, celui qui tout à l'heure s'appelait Toto doit d'appeler Bibi maintenant. On écrit donc la même instruction. Effectivement la seconde boîte de dialogue confirme que le `Singleton` s'appelle maintenant Bibi.

Un modèle de conception est une enveloppe pour des classes correspondant à des problématiques. Vous devrez les compléter pour les adapter au contexte, en y ajoutant les attributs et les méthodes nécessaires. Sinon, vous pouvez utiliser les autres mécanismes de la programmation orientée objet. Par exemple, si vous faites de `Singleton` une classe mère dont d'autres classes peuvent dériver, ces

autres classes n'auront également qu'une seule instance. À vous de voir comment ces modèles s'adaptent le mieux.

Quelques modèles de conception courants

Le nombre des modèles de conception est considérable. Chacun d'entre eux résout une problématique donnée. Dans cette partie, nous survolerons quelques-uns des modèles de conception les plus courants. Nous n'entrerons pas dans les détails et ne les implémenterons pas non plus, car ce n'est pas le sujet de cet ouvrage. Cependant, c'est un aspect important de la programmation, car ce sont des techniques valables, éprouvées, efficaces et fiables.

DisposeFinalize

Vous devriez pour l'instant laisser le ramasse-miettes s'occuper de la libération de la mémoire, car vous risqueriez d'être moins efficace que lui en tentant de forcer vous-même la libération. Effectivement, libérer la mémoire proprement n'est pas chose aisée. L'implémentation du modèle `DisposeFinalize` permet de bien le faire.

Il est composé de plusieurs éléments :

- Il faut tout d'abord un attribut booléen pour déterminer si la destruction a été faite.
- Puis, on doit avoir une fonction `Dispose`. Celle-ci fait appel à une fonction surchargée prenant en paramètre un booléen. Ce booléen détermine si c'est vous qui avez forcé la destruction de l'objet, ou si c'est le ramasse-miettes. La fonction `Dispose` sans paramètre appelle la fonction `Dispose` avec le paramètre `true`, puis la fonction `SuppressFinalize` du ramasse-miettes.
- Ensuite, il faut redéfinir la fonction `Finalize`. Elle appellera la fonction `Dispose` avec le paramètre `false` pour prévenir que c'est le ramasse-miettes qui a lancé la destruction. Si la classe en cours est une classe fille, il faut libérer la classe mère.
- Dans la classe `Dispose` surchargée, selon le paramètre, le traitement de la libération des différents attributs devra être faite.

Si vous implémentez ce modèle, vous pourrez alors détruire vous-même vos objets en faisant appel à la méthode `Dispose`. Elle se chargera de libérer la mémoire dédié à l'objet et aux autres objets qui dépendent de lui.

Iterator

Lorsque les projets sont relativement conséquents, les structures de données complexes, il peut être fastidieux de les parcourir dans leur globalité. En effet, on ne peut pas toujours se contenter d'utiliser une simple boucle `for`, en partant du premier au dernier élément, comme dans les tableaux.

Le modèle `Iterator` permet justement de parcourir des classes structurées complexes, comme si elles n'étaient que de simples listes. De plus, utiliser ce modèle permet de parcourir la structure, mais sans exposer son fonctionnement intérieur. Pour cela, on a besoin de trois éléments :

- Il faut une interface définissant un itérateur. Cette interface contient au moins une méthode permettant d'accéder à l'élément courant, ainsi qu'une méthode permettant d'accéder à l'élément suivant. On peut compléter cette interface en ajoutant la possibilité de supprimer l'élément en cours, mais l'indispensable est de pouvoir accéder à l'élément en cours et au suivant.
- Il faut une classe `MonIterator` adaptée à la structure qui implémente l'interface `Iterator`. Cette classe saura parcourir la structure.
- Dans la classe correspondant à une structure, il faut une méthode qui crée et renvoie un itérateur capable de parcourir cette structure.

Vous pourrez alors parcourir cette structure sans vous soucier de son fonctionnement, étant donné que l'itérateur est fait pour cela, et adapté à ladite structure.

État

Les comportements d'un objet dépendent souvent d'un certain état, à un moment donné. Un magasin par exemple est susceptible d'accueillir des clients s'il est dans un état "*ouvert*". S'il est dans un état "*fermé*", il n'y a pas de clients dedans et les lumières sont éteintes.

La notion d'état détermine donc un certain comportement de l'objet à un moment donné. Le rôle de ce modèle est de symboliser cette notion d'état, sans toucher à l'objet lui-même. De cette manière, on ne rend pas les états dépendant de l'objet. Si par exemple on doit ajouter ou supprimer des états, il ne sera pas nécessaire de remodifier totalement l'objet lui-même.

La méthode communément utilisée pour ce problème est de définir les états comme des constantes (`ETAT_OUVERT`, `ETAT_FERME`), d'avoir un attribut symbolisant l'état en cours, et selon la valeur de cet attribut, de faire les opérations ad hoc. Si le nombre d'états est important, le code peut devenir confus.

Le principe de ce modèle est de définir une interface correspondant à un état. Ensuite, chaque état sera une classe implémentant cette interface.

- Tout d'abord, il faut créer l'interface. Cette interface contiendra l'ensemble des méthodes en rapport avec l'objet, par exemple pour le magasin `servirClient`, `rangerMateriel`, etc., c'est-à-dire l'ensemble des méthodes qui ont une dépendance par rapport à un état donné.
- Ensuite, il faut créer les classes des différents états et leur faire implémenter l'interface. Par exemple, pour l'état `magasinFerme`, la classe retournera une exception, alors que pour `magasinOuvert`, elle effectuera l'opération comme il faut.
- Il faut enfin décrire le mécanisme de changement d'état, permettant de passer de l'un à l'autre et de définir un état de départ.

De cette manière, vous fournissez à l'objet un comportement qui dépend d'un état donné. Or, ce traitement est séparé de l'objet lui-même, ce qui permet de gagner en compréhension, en souplesse et en fiabilité.

L'étude des modèles de conception est bénéfique pour la compréhension des mécanismes de la programmation orientée objet. Nous en avons présenté ici trois qui sont relativement simples. D'autres sont plus complexes et il en existe une multitude, chacun adapté à un problème donné. N'hésitez pas à en utiliser, vos programmes seront de meilleure qualité.

9.5. Quelques bonnes habitudes à prendre

Un bon programme repose sur les habitudes de son programmeur. Il est rare d'avoir un programme de qualité si celui qui l'a conçu l'a fait de manière désinvolte, sans faire attention.

Nous exposerons ici quelques bonnes habitudes à prendre, que vous avez mises en œuvre pour la plupart. Vous comprendrez tout leur intérêt et vous les aurez sous la main.

Pour une meilleure compréhension, bien indenter

L’indentation consiste à commencer des lignes du même bloc au même niveau. On utilise des tabulations pour les décaler. Cela permet une meilleure compréhension dans le code source. La règle est : on entre dans un bloc, on indente, on sort d’un bloc, on revient au niveau de départ. Un bloc peut être la déclaration d’une fonction, une structure de contrôle, etc.

```
Public Sub testIndent(ByVal ind As Boolean)
  If ind
MessageBox.Show("Le paramètre est vrai")
  Else
    MessageBox.Show("Le paramètre est faux")
  End If
End Sub
```

Vous avez ici trois niveaux d’indentation :

- La déclaration de la fonction, alignée à gauche.
- Le contenu de la fonction, qui est décalé d’une tabulation après la déclaration de celle-ci. À ce niveau seront alignées toutes les instructions de la fonction, dont le `If` qui ouvre un nouveau bloc.
- L’intérieur des blocs du `If` et du `Else`, contenant les instructions d’affichage.

L’indentation permet un gain réel de lisibilité. Voici un code sans indentation :

```
Public Sub testIndent(ByVal ind As Boolean)
If ind
MessageBox.Show("Le paramètre est vrai")
Else
MessageBox.Show("Le paramètre est faux")
End If
End Sub
```

Ce bout de programme reste compréhensible car il est petit. Mais il est beaucoup moins lisible ainsi. C’est pourquoi nous vous recommandons de bien indenter vos programmes.

Être clair et expliquer

Soyez explicite. De cette manière, en lisant le code source ultérieurement, vous comprendrez son sens et son intérêt. Prenons le `Singleton` examiné précédemment :

```
Public Class Singleton
  ' Le constructeur est privé,
  ' on ne peut pas instancier la classe
  Private Sub New()
  End Sub

' Instance unique de la classe
' c'est un attribut privé de celle-ci
Private Shared instance As Singleton = Nothing

' L'accesseur crée l'instance si elle n'existe pas
' Puis il la retourne
Public Shared Function getInstance() As Singleton
  If (Me.instance Is Nothin)
    Me.instance = New Singleton()
  End If
  Return Me.instance
End Function

  Public nom As String = "Toto"
End Class
```

Cette classe est claire, explicite et commentée. Grâce aux noms des différents éléments, on comprend à quoi ils correspondent. Vous pouvez vous reporter aux commentaires pour chercher des compléments d'information.

Voici la même classe, sans commentaire, avec des noms qui ne sont pas explicites du tout :

```
Public Class Ljopusd
  Private Sub New()
  End Sub

Private Shared utjvqsd As Ljopusd = Nothing

Public Shared Function oieurIqsg() As Ljopusd
  If (Me.utjvqsd Is Nothing)
    Me. utjvqsd = New Ljopusd()
  End If
  Return Me.utjvqsd
  End Function

  Public pisudgsglqjef As String = "Toto"
End Class
```

On a du mal à s'imaginer que cela est un `Singleton`. Pourtant, c'est exactement la même classe que la précédente, avec un comportement identique. D'où l'intérêt d'être explicite et de mettre des commentaires.

Tester les éléments séparément d'abord

Quand vous testez l'application, faites-le élément par élément. On ne sait jamais, il est possible qu'un défaut quelque part soit compensé par un autre défaut ailleurs. Prenons un cas évident :

```
Public Function addition1(ByVal nbr1 As Integer, _
 ByVal nbr2 As Integer)
  Return nbr1 + nbr2 + 1
End Function

Public Function addition2 (ByVal nbr1 As Integer, _
 ByVal nbr2 As Integer)
  Return nbr1 + nbr2 - 1
End Function
```

Ces deux fonctions effectuent une addition. Elles sont toutes les deux fausses au niveau du sens. Testez-les ainsi :

```
Dim res As Integer
res = addition1(3, 4) + addition2(6, 9)
MessageBox.Show("res = " & res)
```



Figure 9.43 :
Résultat correct avec des fonctions fausses

Vous obtiendrez effectivement le bon résultat car les deux erreurs s'annulent. Pourtant les deux fonctions sont fausses. C'est pourquoi, il faut faire les tests des éléments séparément :

```
MessageBox.Show("somme = " & addition1(3, 4))
MessageBox.Show("somme = " & addition2(6, 9))
```



Figure 9.44 :
Erreur sur addition1



Figure 9.45 :
Erreur sur addition2

De cette manière, vous vous rendrez compte que les deux fonctions sont fausses, et en plus vous pourrez comprendre l'erreur.

Forcer un comportement pour les cas d'erreur

Le pire dans les programmes est l'imprévu. En effet, en tirant parti des cas de figure non traités, les pirates provoquent des bogues, prennent le contrôle sur certaines choses.



Lisez à ce sujet le chapitre ***Rendre un programme robuste.***

```
Dim age As Integer
age = -1
If (age < 18) Then
  MessageBox.Show("Vous êtes mineur")
Else
  MessageBox.Show("Vous êtes majeur")
EndIf
```

Ici, il y a un gros problème au niveau du sens. On est considéré mineur alors que l'âge n'est pas valide du tout. Intégrez donc toujours un mécanisme de vérification de la validité des valeurs, et lorsqu'il y a un problème, définissez un comportement particulier.

```
Dim age As Integer
age = -1
If (age < 0 Or age > 130) Then
  MessageBox.Show("L'âge donné n'est pas valide")
ElseIf (age < 18) Then
    MessageBox.Show("Vous êtes mineur")
Else
    MessageBox.Show("Vous êtes majeur")
EndIf
```

Idem pour les exceptions. Travaillez avec des exceptions sur plusieurs niveaux, de la plus spécialisée à la plus générale, et n'omettez pas les cas d'erreur par défaut :

```
Try
  Dim resultat As Integer
  resultat = dividende \ diviseur
  MessageBox.Show("Résultat = " & resultat)
Catch exc As System.DivideByZeroException
  MessageBox.Show(exc.Message)
  MessageBox.Show(exc.StackTrace)
  diviseur = 1
Catch exc As System.IllegalArgumentsException
  MessageBox.Show(exc.Message)
  MessageBox.Show(exc.StackTrace)
  MessageBox.Show("Les arguments ne sont pas valables")
Catch exc As System.Exception
  MessageBox.Show("Il y a eu une exception d'ordre général")
End Try
```

Les recommandations peuvent être nombreuses, par exemple toujours écrire une seule classe dans un même fichier. Mais si vous appliquez celles-ci à la lettre, vos programmes seront fiables et agréables à utiliser. Par ailleurs, il vous sera plus facile de vous replonger dans votre code après de longs temps d'interruption.

9.6. Bien dissocier les parties de votre programme

Depuis le début, nous sommes partis du principe que vous étiez le seul à créer un projet. Chacun des programmes et des exemples qui ont été présentés était totalement indépendant et fournissait une réponse à un besoin, sans aucune interaction avec un quelconque élément externe.

Or, dans votre vie de programmeur, ce ne sera certainement pas toujours le cas. C'est pourquoi il faut avoir à l'esprit que vos travaux pourront être utilisés par d'autres. Il faut donc prendre des mesures et vous arranger pour rendre vos programmes le plus réutilisables possible.

Pour développer ce point, nous allons voir une application indépendante qui, même si elle répond à une demande de manière efficace, peut provoquer un problème au moment d'interagir avec un autre

programme. C'est pourquoi nous verrons dans un second temps comment modifier ce même programme, de manière à le rendre réutilisable.

C'est dans cet esprit qu'il vous faudra créer vos futurs programmes, car cela n'enlève en rien les possibilités que vous avez précédemment produites, mais en les rendant réutilisables vous leur donnez une valeur ajoutée qui sera grandement appréciée par vos collaborateurs.

Une application indépendante

Pour montrer les limites d'une application indépendante où tout est codé dans le même projet, créons une application complète dans un unique projet.

Imaginons une application qui donne les futurs résultats du loto. Vous conviendrez que beaucoup de monde peut être attiré par cette perspective. Malheureusement, la formule magique n'existant pas, nous allons créer un générateur de six nombres aléatoires. Avec un peu de chance, parmi les essais que vous ferez, vous trouverez les bons.

1 Créez un nouveau projet Application Windows.

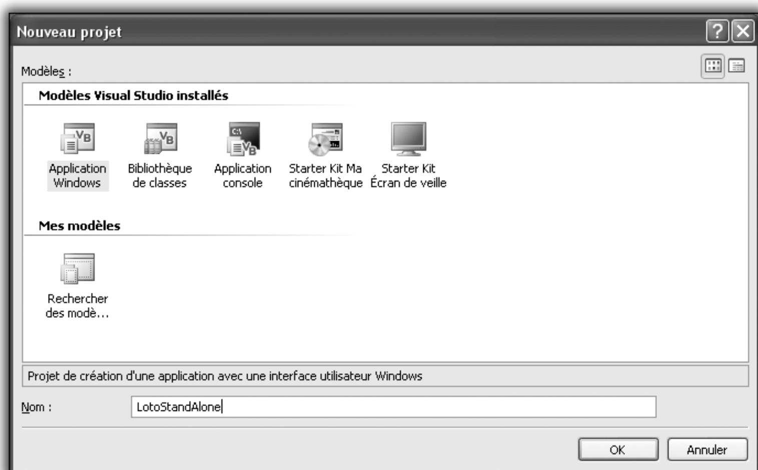


Figure 9.46 : *Nouvelle application Windows*

2 Créez maintenant une petite interface permettant de visualiser vos résultats.



Figure 9.47 :
Création de l'interface

Créons ensuite le code principal de votre programme. Nous allons lui faire créer sept numéros entre 1 et 49 que nous afficherons dans cette interface.

Générer un nombre aléatoire est très facile grâce au framework .Net. Il suffit d'instancier un objet de la classe `Random` et de faire appel à la méthode `Next`. Cette dernière prend deux entiers en paramètres correspondant aux limites basse et haute. Dans notre cas, ces paramètres sont 1 et 49, les limites du Loto.

3 Double-cliquez sur le bouton afin que Visual Studio crée la méthode destinée à gérer l'événement du `Click`. Vous pourrez ensuite y mettre votre méthode.

```
 Private Sub Button1_Click(ByVal sender As System
✂ .Object, _
      ByVal e As System.EventArgs) Handles
     ✂ Button1.Click
    Dim generator As New Random()

    Dim number As Integer = generator.Next(1, 49)
    TextBox1.Text = number
    number = generator.Next(1, 49)
    TextBox2.Text = number
    number = generator.Next(1, 49)
   TextBox3.Text = number
    number = generator.Next(1, 49)
    TextBox4.Text = number
    number = generator.Next(1, 49)
    TextBox5.Text = number
    number = generator.Next(1, 49)
    TextBox6.Text = number
    number = generator.Next(1, 49)
```

```
            TextBox7.Text = number
        End Sub
```

Vous avez maintenant un programme capable de vous donner un tirage du loto. Si un nombre apparaît plusieurs fois, cliquez de nouveau pour avoir un nouveau tirage.

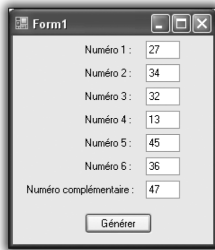


Figure 9.48 :
Tirage du loto

De mon côté, j'ai également une application. Pour l'instant, elle ne fait pas grand-chose, mais je ne désespère pas de pouvoir l'enrichir.

```
Module Module1

    Sub Main()
        Console.WriteLine("Bonjour le monde.")
        Console.ReadLine()
    End Sub

End Module
```

À ce stade, tout cela n'est pas d'une grande utilité. J'y inclurais bien la possibilité de générer un tirage du loto. De cette manière, quelqu'un qui utiliserait mon application pourrait voir "*Bonjour le monde.*" mais également voir le tirage du loto, qui pourrait lui amener la richesse.

Cela tombe bien, vous avez réalisé une application capable de faire ceci.

Quel malheur, c'est une application indépendante et je ne peux donc pas l'intégrer à mon projet personnel. Mon application est condamnée à afficher simplement "*Bonjour le monde.*".

La même application réutilisable

La clé pour pouvoir réutiliser une application est déjà de bien discerner les grandes parties qui la composent, et comment elle pourrait être divisée.

La vôtre pourrait se diviser comme ceci :

1 La partie qui génère les nombres.
2 La partie qui les affiche.

Pour l'instant, tout est au même endroit, et c'est cela qu'il faudrait modifier pour rendre cette application réutilisable.

Le générateur de tirage

Nous avons identifié comme une grande partie de votre programme celle qui fait le calcul des tirages.

Pour l'instant, il n'est aucunement question d'affichage ici, donc, une bibliothèque de classe nous permet de gérer cette partie.

1 Créons donc un nouveau projet bibliothèque de classe.

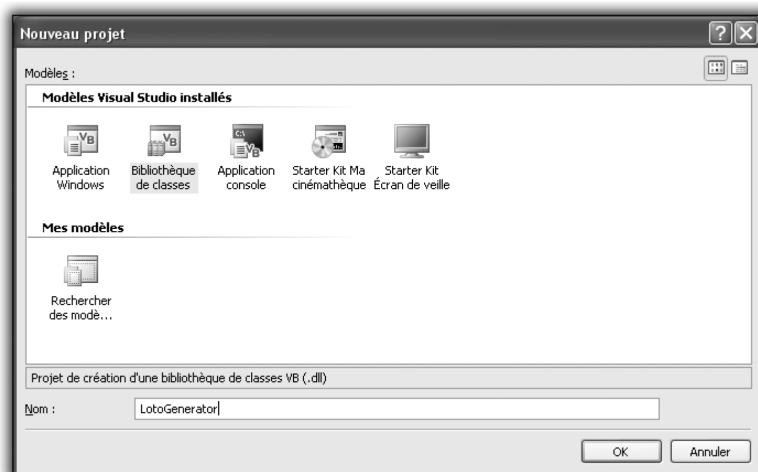


Figure 9.49 : *Nouvelle bibliothèque de classe*

Il nous faudra maintenant remplir cette bibliothèque avec de nouvelles classes.

2 Créons donc une classe qui va générer nos nombres.

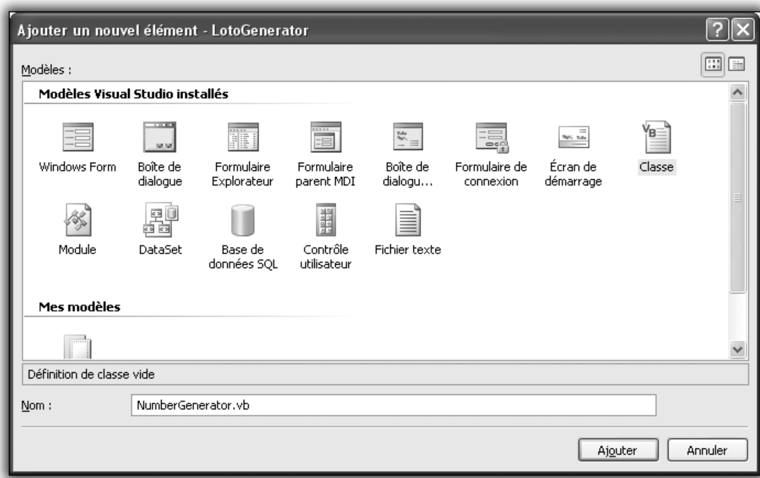


Figure 9.50 : *Nouvelle classe NumberGenerator*

Puis il va falloir remplir cette classe afin qu'elle fasse bien le traitement que l'on veut. C'est là qu'on doit faire preuve d'un peu de réflexion pour savoir comment gérer ceci.

Tout d'abord, posons-nous la question : "*De quoi a-t-on besoin ?*"

Réponse : rien ! En effet, nous avons toutes les informations nécessaires. Notre but étant simplement de générer sept nombres compris entre 1 et 49. Nous n'aurons donc pas d'attributs membres.

Autre question, maintenant : "*Que va-t-on faire ?*"

Notre but est de générer sept nombres que l'on va pouvoir utiliser. Faisons donc une méthode qui renvoie une liste de nombres. Ce qui amène la question suivante : "*Quels seront les paramètres de cette méthode ?*"

Là encore, aucun. Nous savons que ces nombres seront compris entre 1 et 49.

Dernière question pour compléter la création de la méthode : "*A-t-on besoin d'une instance particulière ?*" Réponse : non. Cette méthode étant totalement utilitaire, elle ne possède aucune information pertinente que l'on ait envie de garder. Dans ce cas, on peut la marquer `Shared`, ce qui nous permettra d'accéder à la méthode sans avoir besoin d'instancier un objet de type `NumberGenerator`.

Ce qui nous donne la signature suivante pour cette méthode :

```
Public Shared Function GenerateNumbers() As List(Of
✂ Integer)

End Function
```



Classes et méthodes Public

Pour être utilisées par d'autres, la classe et la méthode devront être `Public`.

Nous avons maintenant fait le plus difficile. Le remplissage de la méthode est alors basique. Il vous suffit de créer le résultat et de le remplir en utilisant ce que vous avez fait précédemment dans l'autre programme.

```
Public Shared Function GenerateNumbers() As List(Of
✂ Integer)
    Dim result As List(Of Integer) = New List(Of
    ✂ Integer)(7)
    Dim i As Integer
    Dim generator As Random = New Random()
    For i = 1 To 7
        result.Add(generator.Next(1, 49))
    Next
    Return result
End Function
```

Parfait, nous venons de terminer la première partie de votre programme. La compilation de ce projet permettra de générer une DLL qui contiendra la classe `NumberGenerator`, qui expose une méthode `GenerateNumbers`.

Occupons-nous maintenant de la seconde partie, l'affichage. Comme nous ne voulons pas modifier notre bibliothèque de classe, ajoutons un projet "*Application Windows*" qui se chargera d'afficher les résultats obtenus depuis notre classe `NumberGenerator`.

En fait, reprenons l'ancien projet : `LotoStandAlone`. Nous gardons l'interface, mais supprimons tous les traitements qui sont faits.

```
Private Sub Button1_Click(ByVal sender As System
✂ .Object, _
    ByVal e As System.EventArgs) Handles Button1.Click

End Sub
```

Or c'est maintenant notre DLL `LotoGenerator` qui fait tous les traitements. Il faudra donc l'inclure au projet.

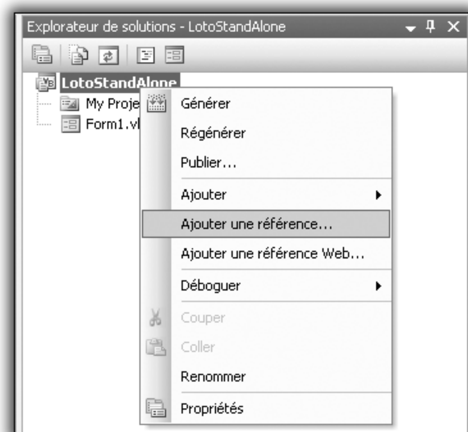


Figure 9.51 : *Nouvelle référence*

Cherchez la DLL, qui doit être dans le dossier d'exécution du projet `LotoGenerator`.

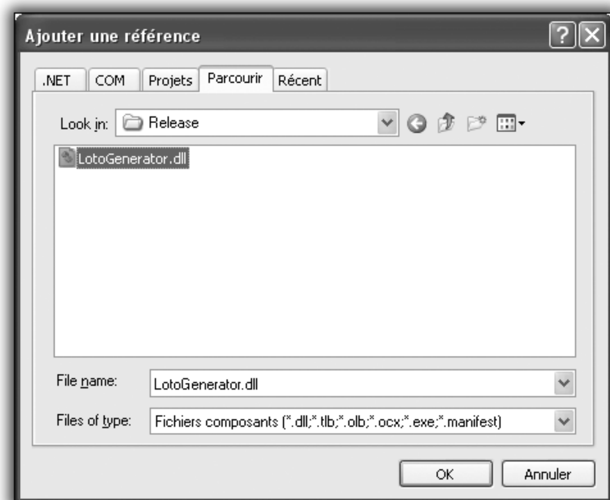


Figure 9.52 : *Inclusion de la DLL*

Pour pouvoir utiliser les classes de la DLL, il vous faut inclure son *namespace*.

```
Imports LotoGenerator
```

Maintenant, il faut appeler la méthode `GenerateNumbers` et adapter l'affichage à partir du résultat.

```
Private Sub Button1_Click(ByVal sender As System
✂ .Object, ByVal e As System.EventArgs) Handles Button1
✂ .Click
    Dim tirage As List(Of Integer) = NumberGenerator
   ✂ .GenerateNumbers()
    TextBox1.Text = tirage.Item(0)
    TextBox2.Text = tirage.Item(1)
    TextBox3.Text = tirage.Item(2)
    TextBox4.Text = tirage.Item(3)
    TextBox5.Text = tirage.Item(4)
    TextBox6.Text = tirage.Item(5)
    TextBox7.Text = tirage.Item(6)
End Sub
```

Vous remarquez qu'on distingue directement les deux parties du programme. Une instruction fait la récupération des nombres, le reste des instructions sert à afficher. Et, finalement, l'application est exactement la même.

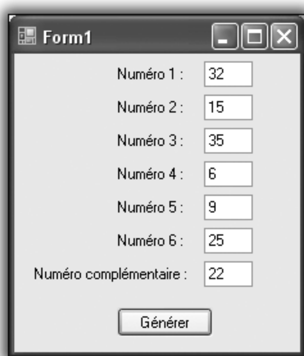


Figure 9.53 :
Application finale identique

De votre point de vue, cela n'a effectivement rien changé. Maintenant, en ce qui me concerne, je peux enrichir mon application qui faisait "Bonjour le monde.".

Il me suffit d'ajouter votre DLL à mon projet et d'ajouter le *namespace*, comme vous l'avez fait dans la partie affichage. Et je peux maintenant utiliser votre méthode de génération de nombres. Puis j'adapte mon propre affichage pour qu'il soit en accord avec mon application :

```
Sub Main()
    Console.WriteLine("Bonjour le monde.")
```

```
        Dim tirage As List(Of Integer) = NumberGenerator
      ✂ .GenerateNumbers()
        Console.WriteLine(tirage.Item(0))
        Console.WriteLine(tirage.Item(1))
        Console.WriteLine(tirage.Item(2))
        Console.WriteLine(tirage.Item(3))
        Console.WriteLine(tirage.Item(4))
        Console.WriteLine(tirage.Item(5))
        Console.WriteLine(tirage.Item(6))
        Console.ReadLine()
    End Sub
```

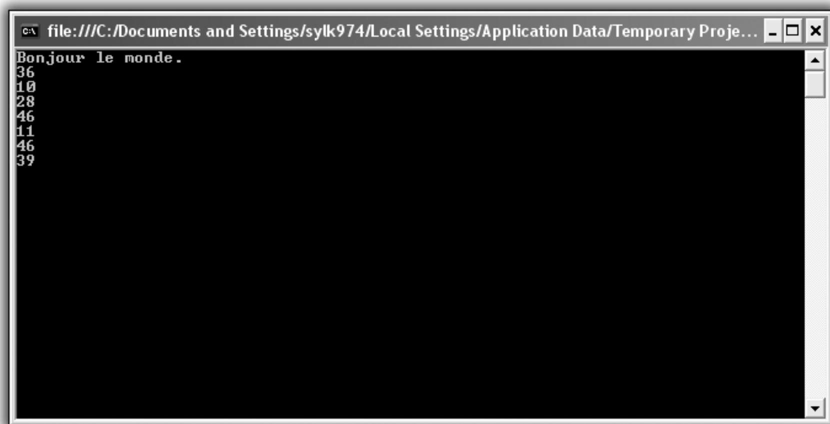


Figure 9.54 : *Ma version du tirage du loto*

Et, grâce à votre DLL, j'ai pu enrichir mon application. De votre côté, vous avez rendu votre travail réutilisable, ce qui lui a donné une énorme valeur ajoutée. Si vous aviez gardé la version d'origine, où tout est fait au même endroit (traitement + affichage), je n'aurais pu l'utiliser, et je ne l'aurais certainement jamais fait. Alors qu'en la découpant comme il faut, en la divisant en deux parties distinctes et dissociées, vous m'avez permis de l'utiliser dans une application totalement extérieure à votre projet.

Cette capacité à bien discerner les parties d'un projet demande de l'expérience et de la pratique. Il n'est pas intuitif de faire un découpage qui n'a pas lieu d'être au premier abord, surtout dans une optique où vous avez le contrôle total de votre application. Qui plus est, cela demande plus de temps, car ce sont autant d'éléments à prendre en compte, en plus de ceux qui existent déjà. Mais, pour deux applications

qui sont identiques visuellement, la réutilisabilité peut faire la différence entre la très bonne application et celle qui est bien, tout simplement.

9.7. Utiliser de bons outils pour de bons résultats

Pour réaliser un bon programme, produire du code de bonne qualité est une chose, mais ce n'est pas l'unique qualité que peut avoir votre programme. La capacité à suivre son déroulement ou encore la manière dont il est documenté sont autant de points positifs pour faire un bon programme.

En effet, vous pouvez avoir fait le meilleur code du monde, si vous êtes le seul à pouvoir le lire il y a de fortes chances qu'il tombe aux oubliettes. Si tout le temps qu'est censé faire gagner votre programme doit être utilisé pour le comprendre ou pour analyser son comportement en cas de problème, son intérêt est totalement perdu.

Cependant, il ne vous est pas forcément nécessaire de vous occuper de toutes ces étapes vous-même. Fort heureusement, d'autres l'ont fait avant vous et, honnêtement, il serait dommage de ne pas en profiter.

Parmi ces outils, nous allons nous attarder sur quelques outils de référence qui vous feront gagner un temps précieux et amélioreront fortement la qualité de votre projet, sans que vous ayez pour autant beaucoup de code à produire.

Logger avec log4net

Dans la partie des gestions d'erreurs, nous avons vu comment créer des journaux d'événements qui permettent de garder une trace d'un comportement particulier de votre programme.



*Reportez-vous à ce sujet au chapitre **Création de journaux d'événements**.*

Or il n'y a pas que les erreurs qui sont bonnes à être enregistrées. Avoir une vue directe sur les arguments passés à vos méthodes ou encore savoir quelles méthodes votre programme exécute sont autant

d'informations essentielles qui vous permettront de comprendre rapidement ce qui s'est passé dans votre programme et donc de réagir en conséquence.

Un outil de référence nous permet de gérer ceci : Log4Net.

Présentation de l'outil

Log4Net est un outil qui vous permettra de gérer le déroulement de votre programme à travers des "logs", c'est-à-dire des fichiers textuels qui détailleront le comportement de votre application, un peu à la manière des journaux d'événements, qui permettent d'enregistrer les erreurs.

Un des aspects intéressants de cet outil est sa capacité à être configuré à l'exécution. De cette manière, vous n'avez pas à modifier votre programme ni à le recompiler avec des logs plus ou moins détaillés. Il offre de nombreuses possibilités telles les suivantes :

- **La compatibilité avec de nombreux frameworks**. Il est possible de l'exécuter avec la plupart des versions du framework .Net, même celle destinée aux téléphones portables, ou encore Mono, l'implémentation de .Net portable sur d'autres systèmes d'exploitation, comme Linux.
- **La multiplicité des sorties possibles**. Les informations recueillies par Log4Net peuvent vous être restituées sous forme de fichiers textuels, c'est le fonctionnement normal. Mais vous pouvez également les retranscrire sur la Console, ou encore envoyer des courriers électroniques, ou tout simplement les garder en mémoire…
- **La configuration dynamique**. Log4Net offre la possibilité d'être configuré à travers un fichier XML. De cette manière, sans modifier votre application, vous pouvez changer le niveau de log que vous voulez voir afficher, la sortie que vous souhaitez utiliser ou encore le chemin qui contiendra votre fichier de log.
- **Le contexte de log**. Outre les informations que vous avez explicitement demandées, Log4Net enregistre un certain nombre d'informations qui lui sont propres, comme l'heure à laquelle il a effectué le traitement ou encore le thread ayant appelé l'opération.
- **Une hiérarchie de criticité**. Nous sommes d'accord pour dire que toutes les informations n'ont pas la même importance. Savoir que

le programme est entré dans telle méthode est moins important qu'une erreur ayant surgi sans prévenir. Pour marquer cette importance, Log4Net définit plusieurs niveaux dans les logs :

- *Debug*. Ce sont les informations les moins importantes. On pourra y mettre les entrées et les sorties de méthodes ou encore des messages informatifs pour nous dire dans quelle partie du programme nous sommes passés.
- *Info*. Ce niveau de log correspond à une information pertinente. Cela pourra correspondre à une action importante ou encore à l'affichage des données utilisées dans notre méthode.
- *Warn*. Ce niveau marque un avertissement. Un avertissement n'est pas une erreur en soi, mais marque un état qui peut provoquer un comportement instable. Il n'y a pas lieu de s'affoler, mais il est quand même bon de savoir que quelque chose de troublant s'est passé.
- *Error*. C'est le dernier niveau de log utilisé, et le plus important. Il est généralement utilisé en cas d'erreur. En effet, étant donné qu'une erreur n'est jamais faite volontairement, il ne faut surtout pas la rater. En loggant vos erreurs avec ce niveau, vous êtes certain de les voir apparaître le cas échéant.

Après cette rapide présentation de l'outil, entrons dans le vif du sujet.

Utilisation dans vos applications

Pour pouvoir utiliser Log4Net, il faut tout d'abord vous le procurer. Ce projet est un projet Open Source, donc totalement ouvert à l'utilisation, et ceci gratuitement. Vous pouvez l'obtenir sur le site **http://logging.apache.org/log4net/index.html**.

Une fois que vous aurez récupéré le package de Log4Net, il vous faudra prendre la bonne version ; pour nous, la DLL compatible avec la version du framework .Net 2.0. Après avoir décompressé l'archive de Log4Net, vous prendrez la DLL `log4net.dll`, qui se trouve dans le dossier *bin/net/2.0/release*.

Pour pouvoir utiliser Log4Net dans vos projets dès maintenant, vous devez l'inclure dans vos applications. Pour cela, ajoutez la DLL dans les références de vos projets.

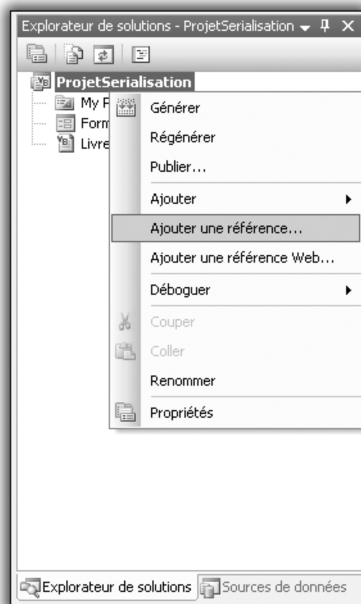


Figure 9.55 : *Ajout d'une référence*

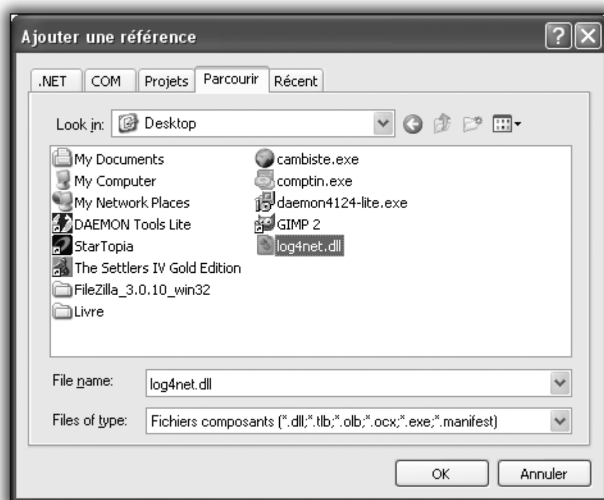


Figure 9.56 : *Chercher la DLL*

Une fois que la référence est ajoutée, dans chaque fichier de votre projet où vous voudrez utiliser Log4Net vous inclurez le namespace `log4net` et `Config`.

```
Imports log4net
Imports log4net.Config
```

Une fois toutes ces étapes réalisées, vous pourrez utiliser Log4Net dans vos projets.

À titre d'exemple, vous pouvez créer une application légère contenant cette méthode, en ayant bien effectué avant les opérations décrites ci-dessus.

```
Module Module1

    Sub Main()
        Dim log As ILog = LogManager
      ✂ .GetLogger(GetType(Module1))

        BasicConfigurator.Configure()
        log.Info("Entering application.")
        log.Info("Exiting application.")

        Console.WriteLine("Action finished.")
        Console.ReadLine()
    End Sub

End Module
```

Voici le résultat de ce bout de programme :

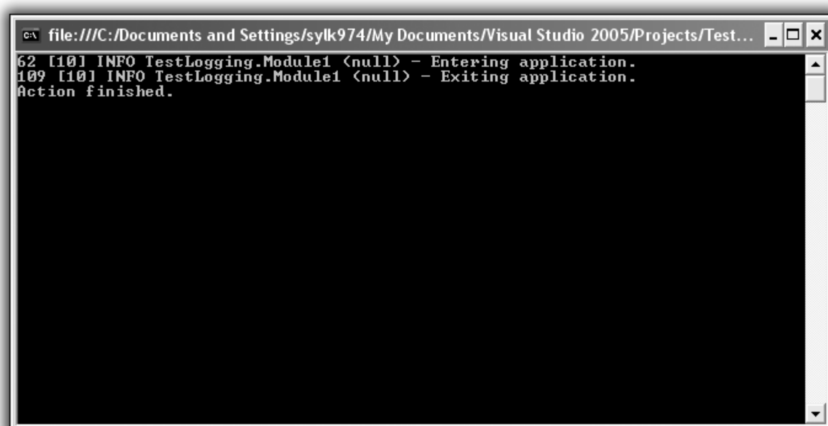


Figure 9.57 : *Premier programme utilisant Log4Net*

On ne dirait pas, mais dans ce bout de programme nous avons fait pas mal de choses.

```
Dim log As ILog = LogManager.GetLogger(GetType(Module1))
```

Cette ligne crée le `logger`. C'est avec cet objet que vous allez effectuer tous vos traitements de log. Le paramètre de la méthode `GetLogger` est le type de la classe ou du module utilisé. Ce paramètre n'est pas obligatoire, mais il est possible de faire apparaître cette information dans les logs, c'est pourquoi il est bon de la donner au moment d'instancier le `logger`.

```
BasicConfigurator.Configure()
```

En instanciant le `logger` sans faire d'autre action, celui-ci ne fera rien. En effet, pour que ses actions aient une répercussion visible par l'utilisateur, il faut le configurer en conséquence. Comme nous n'avons pas encore vu les détails concernant la configuration d'un `logger`, nous utilisons la configuration par défaut, accessible *via* la classe `BasicConfigurator`.

```
log.Info("Entering application.")
log.Info("Exiting application.")
```

Voici comment on utilise le `logger`. Il suffit tout simplement d'appeler la méthode correspondant au niveau de log que l'on veut donner et de passer en paramètre le message que l'on veut voir apparaître.

Et, comme nous l'avons vu après l'exécution de notre application, la configuration par défaut renvoie en fait les messages de log sur la `Console`, nous permettant ainsi de suivre ce qui se passe sans avoir à écrire d'instruction `Console.WriteLine`.

Dans la sortie de notre programme, vous avez pu remarquer que l'on trouve un certain nombre d'informations complémentaires qui nous sont fournies par Log4Net.

Nous trouvons entre autres le niveau de log utilisé (`INFO`) ou encore le module qui a exécuté l'action (`TestLogging.Module1`). Les premiers nombres sont des informations de temps et le thread dans lequel l'instruction a été faite.

Maintenant, nous allons créer une nouvelle méthode et placer le `logger` à un endroit accessible par toutes les méthodes.

```
Module Module1
    Dim log As ILog = LogManager
   ✂ .GetLogger(GetType(Module1))

    Sub HelloWorld()
```

```
        log.Debug("Entering Hello World method")
        Console.WriteLine("Hello world.")
        log.Debug("Leaving Hello World method")
    End Sub

    Sub Main()
        BasicConfigurator.Configure()
        log.Info("Entering application.")
        HelloWorld()
        log.Info("Exiting application.")

        Console.WriteLine("Action finished.")
        Console.ReadLine()
    End Sub

End Module
```

Voici maintenant ce que l'on peut observer :

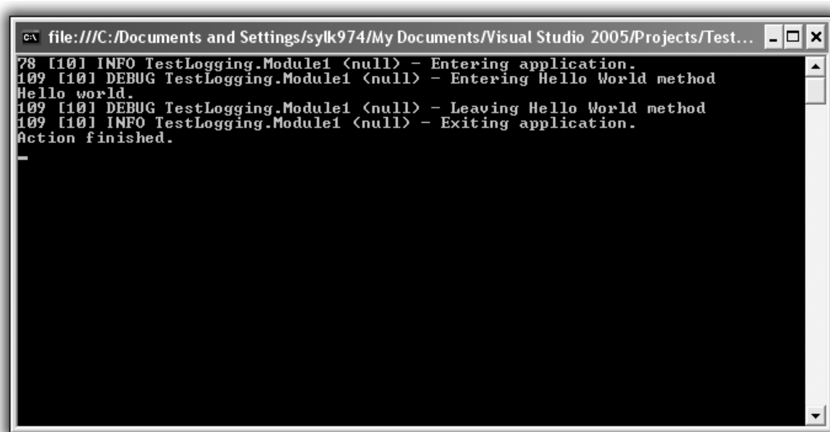


Figure 9.58 : *Bonjour le monde !*

Nous pouvons voir l'exécution de la méthode `HelloWorld`, ainsi que l'affichage de toutes les informations de log qui y sont associées. Nous pouvons en déduire que la configuration par défaut affiche les logs jusqu'au niveau le plus bas.

Très bien, nous pouvons maintenant enrichir nos applications d'informations pertinentes qui seront affichées sur la Console et qui nous permettront de suivre notre programme. En revanche, et je pense que vous serez d'accord avec moi, on s'y perd un peu. Notre message "*Hello World*", correspondant au réel traitement de notre méthode, passe

un peu inaperçu à cause de la pollution visuelle provoquée par tous ces messages de log.

Pour éviter ceci, nous allons configurer nous-mêmes le `logger`. La configuration d'un `logger` se fait en utilisant le format XML. Nous la ferons dans un fichier nommé *Logger.config*. Pour ce faire, nous allons remplacer le `BasicConfigurator` par un `XmlConfigurator` qui prend en paramètre un `FileInfo` lié à notre fichier *Logger.config.*

```
XmlConfigurator.Configure(New System.IO.FileInfo("Logger
✂ .config"))
```

Voyons maintenant comment écrire un fichier de configuration du `logger`.

Tout d'abord, il faut créer le squelette, c'est-à-dire une balise `log4net` qui contiendra toutes les informations du `logger`.

```
<log4net>
</log4net>
```

Comment configurer un `logger` ?

En fait, Log4Net fonctionne avec un système d'`Appender`.

Un `Appender` est une manière de restituer ce qui a été traité par le `logger`. Par exemple, l'écriture sur la Console correspond à un `Appender`, l'écriture dans un fichier correspond à un autre type d'`Appender`. Il faudra donc marquer dans le fichier de configuration tous les `Appender` que votre `logger` devra traiter. Il est défini par un nom, un type et un format de sortie. Pour les détails à propos des formats de sortie, je vous invite à chercher dans la documentation de Log4Net. Pour notre application, nous utiliserons le format de sortie par défaut.

```
  <appender name="A1" type="log4net.Appender
 ✂ .ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
     ✂ %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>
```

Maintenant que nous avons défini un `Appender`, il faut que nous précisions que le `logger` utilise cet `Appender`. En utilisant la balise `root`, nous définissons une configuration globale pour tous les `logger`.

Dans cette balise, il nous faut également définir le niveau minimal d'affichage des logs.

```
<root>
  <level value="DEBUG" />
  <appender-ref ref="A1" />
</root>
```

Voici donc notre fichier de configuration au complet :

```
<log4net>
  <appender name="A1" type="log4net.Appender
 ✂ .ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
     ✂ %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>
  <root>
    <level value="DEBUG" />
    <appender-ref ref="A1" />
  </root>
</log4net>
```

En exécutant de nouveau notre programme, nous retrouvons notre sortie d'origine sauf que, cette fois-ci, nous avons utilisé un fichier de configuration.

Tentons de changer le niveau de log, en mettant `INFO` à la place de `DEBUG`.

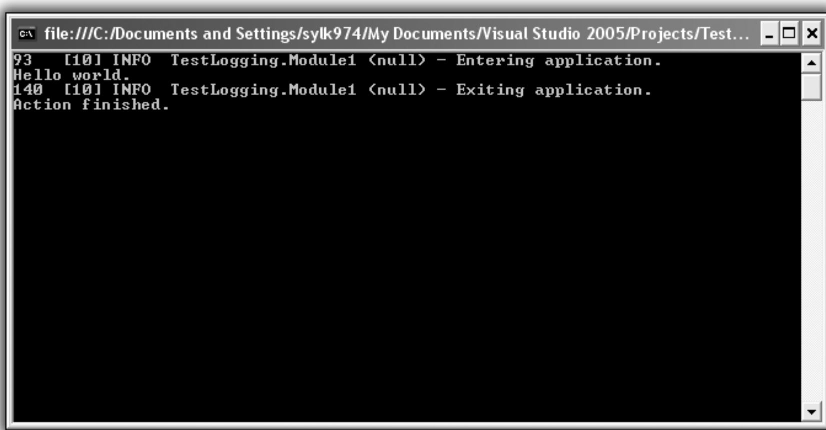


Figure 9.59 : *Niveau de log INFO*

Cela devient un peu plus lisible. Et nous avons pu faire ceci sans retoucher à notre programme.

Très bien mais, pour l'instant, nous affichons toujours les logs sur la Console. Nous allons maintenant tenter d'afficher nos logs dans un fichier.

Pour cela, il faut ajouter un `Appender` dans notre fichier de configuration : le `FileAppender`. Il faut également y définir le nom du fichier de sortie, le format de sortie, mais également si on écrase les anciens fichiers ou non. Voici ce que l'on doit ajouter à notre fichier de configuration :

```
  <appender name="FileAppender" type="log4net.Appender
 ✂ .FileAppender">
    <file value="log.txt" />
    <appendToFile value="false" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
     ✂ %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>
```

Nous avons ajouté notre nouvel `Appender`, il faut donc maintenant préciser dans `root` que l'on veut utiliser celui-ci :

```
  <root>
    <level value="DEBUG" />
    <appender-ref ref="FileAppender" />
  </root>
```

Le fichier au complet donne ceci :

```
<log4net>
  <appender name="A1" type="log4net.Appender
 ✂ .ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
       %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <appender name="FileAppender"
            type="log4net.Appender.FileAppender">
    <file value="log.txt" />
    <appendToFile value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp
         [%thread] %-5level %logger %ndc -
        ✂ %message%newline" />
```

```
  </layout>
  </appender>
  <root>
    <level value="DEBUG" />
    <appender-ref ref=" FileAppender " />
  </root>
</log4net>
```

Voici ce que l'on obtient en exécutant de nouveau notre programme :

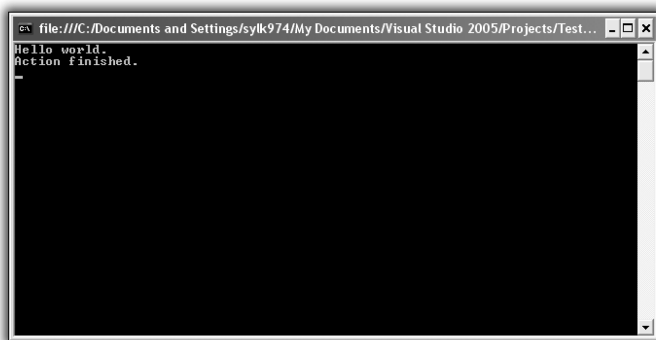


Figure 9.60 : *Logging dans un fichier*

La Console n'est maintenant plus polluée par tous les messages de Log4Net.

A-t-on perdu toutes nos informations pour autant ? Que nenni ! Si nous allons dans le dossier de notre projet, nous pouvons remarquer la création d'un nouveau fichier *Log.txt* (comme le nom que nous avons donné à notre fichier de sortie). Et qu'y a-t-il dedans ?

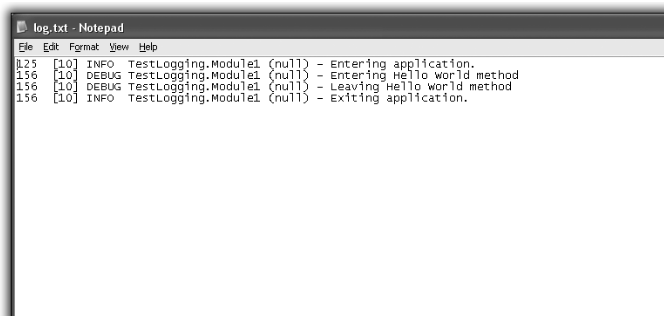


Figure 9.61 : *Fichier de sortie*

Il y a toutes nos informations de logs. Grâce à ce fichier de log, nous pouvons maintenant tracer *a posteriori* ce qui a été réalisé par notre programme.

Le `FileAppender` est la forme la plus simple d'`Appender` sur les fichiers. Or Log4Net possède un autre `Appender` pour les fichiers : le `RollingFileAppender`. Cet `Appender` permet d'utiliser plusieurs fichiers de logs de taille plus petite. En effet, selon votre niveau de logs et la manière dont vous les utilisez dans votre programme, un fichier de log peut très vite devenir énorme.

Essayons de répéter `HelloWorld` 100 000 fois.

```
Sub Main()
    XmlConfigurator.Configure(New System.IO.FileInfo _
        ("Logger.config"))
    log.Info("Entering application.")
    Dim i As Integer
    For i = 0 To 100000
        HelloWorld()
    Next
    log.Info("Exiting application.")

    Console.WriteLine("Action finished.")
    Console.ReadLine()
End Sub
```

Le programme prend un peu de temps, mais allons voir ce qui s'est passé du côté de notre fichier.

Il fait maintenant près de trente mégaoctets. C'est énorme ; si vous essayez de l'ouvrir avec un éditeur de texte, vous risquez d'avoir un peu de mal. Le `RollingFileAppender` permet de scinder un fichier de log en plusieurs fichiers plus légers.

Comme pour tous les `Appender`, il faut lui définir un nom et un format de sortie. Comme il s'agit d'un `Appender` sur fichiers, il faut également préciser le nom du fichier et si on écrase les anciens. De plus, pour le `RollingFileAppender`, il va falloir préciser la taille maximale d'un fichier et le nombre maximal de fichiers.



Taille des fichiers Log

Si l'ensemble des logs dépasse la taille maximale pour tous les fichiers, les dernières informations seront perdues.

```
<log4net>
  <appender name="A1" type="log4net.Appender.ConsoleAppender">
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
         %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <appender name="FileAppender"
            type="log4net.Appender.FileAppender">
    <file value="log.txt" />
    <appendToFile value="true" />
    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
         %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <appender name="RollingFile"
            type="log4net.Appender.RollingFileAppender">
    <file value="log.txt" />
    <appendToFile value="true" />
    <maximumFileSize value="10000KB" />
    <maxSizeRollBackups value="2" />

    <layout type="log4net.Layout.PatternLayout">
      <conversionPattern value="%-4timestamp [%thread]
         %-5level %logger %ndc - %message%newline" />
    </layout>
  </appender>

  <root>
    <level value="DEBUG" />

    <appender-ref ref="RollingFile" />
  </root>
</log4net>
```

Si vous ouvrez maintenant le dossier de votre projet, vous remarquez un fichier *Log.txt*, mais également un fichier *Log.txt.1*. Ce dernier fichier est en fait le premier fichier de log, qui étant devenu trop volumineux a laissé sa place à un autre. De cette manière, pour les analyser, vous pouvez regarder des fichiers d'une taille que vous aurez choisie.

Avec ce que nous avons vu au cours de ce chapitre, vous pouvez renforcer vos applications avec une sorte de mouchard capable de vous dire tout ce qu'il fait, quand il le fait, et avec quoi il l'a fait. Cependant, Log4Net expose infiniment plus de possibilités que celles que nous

avons vues. N'hésitez pas à regarder la documentation de Log4Net et à faire toutes sortes de tests et d'essais pour connaître son comportement.

Avoir une documentation professionnelle : Ndoc

Au risque de me répéter, j'aimerais encore appuyer sur le fait qu'il est important de bien commenter votre code et de le documenter, de manière qu'une autre personne qui utilise vos travaux puisse les comprendre. En effet, vous n'allez peut-être pas toujours être tout seul à travailler sur une application et, si le projet commence à être un peu important, on peut vite s'y perdre.



*Reportez-vous à ce sujet au chapitre **Quelques bonnes habitudes à prendre**.*

Admettons que vous veniez de terminer un travail : vous avez créé une DLL capable de faire des opérations mathématiques. Vous proposez à qui veut bien s'en servir la possibilité de faire des opérations basiques sur des nombres. Voici ce qu'une personne qui possède le code pourra voir :

```
 Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
     Return x + y
 End Function

 Public Function Substract(ByVal x As Integer, ByVal y
✂ As Integer) As Integer
     Return x - y
 End Function

 Public Function Times(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
     Return x * y
 End Function

 Public Function Divide(ByVal x As Double, ByVal y As
✂ Double) As Double
     Return x / y
 End Function
```

Quelqu'un qui ne connaît pas du tout le Visual Basic peut se retrouver perdu. Pour aider ces personnes, il est judicieux d'ajouter quelques commentaires qui permettront de mieux comprendre les méthodes.

```
Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
    ' Retourne l'addition entre x et y
    ' Cette opération se fait sur nombres entiers
    Return x + y
End Function

Public Function Substract(ByVal x As Integer, ByVal y
✂ As Integer) As Integer
    ' Retourne la soustraction entre x et y
    ' Cette opération se fait sur nombres entiers
    Return x - y
End Function

Public Function Times(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
    ' Retourne la multiplication entre x et y
    ' Cette opération se fait sur nombres entiers
    Return x * y
End Function

Public Function Divide(ByVal x As Double, ByVal y As
✂ Double) As Double
    ' Retourne la division entre x et y
    ' Cette opération se fait sur nombres décimaux
    ' En effet, le résultat d'une division n'étant pas
   ✂ exact,
    ' il est préférable de travailler avec des Double
    Return x / y
End Function
```

Cette étape est déjà pertinente, car de cette manière quelqu'un qui ne connaît pas le Visual Basic peut lire les commentaires et comprendre le résultat des méthodes. De plus, ces commentaires expliquent pourquoi l'addition, la multiplication et la soustraction travaillent sur des nombres entiers, alors que la division travaille avec des nombres décimaux.

Or, dans ce cas, il faut absolument que cette personne ait le code de votre application pour comprendre ce que vous avez voulu faire.

Il existe cependant un moyen de produire une documentation externe à votre programme, de manière à pouvoir l'exploiter sans posséder le code.

La documentation XML

Le framework .Net et Visual Studio permettent de produire cette documentation en utilisant le format XML. Il faudra alors définir pour chacune de vos méthodes une portion XML correspondant à votre documentation. Écrivez trois apostrophes ("’") au-dessus d’une de vos méthodes :

```
''' <summary>
'''
''' </summary>
''' <param name="x"></param>
''' <param name="y"></param>
''' <returns></returns>
''' <remarks></remarks>
Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
    ' Retourne l'addition entre x et y
    ' Cette opération se fait sur nombres entiers
    Return x + y
End Function
```

Visual Studio va ajouter automatiquement des balises XML vous permettant de décrire vos applications. Il ne vous restera alors plus qu’à la remplir.

- `Summary` donne une description générale de la méthode. Vous pourrez y décrire le traitement effectué par votre méthode.
- `Param` est la balise qui vous permettra de décrire les arguments de votre méthode. Le nom du paramètre est déjà marqué pour vous permettre de savoir quel argument vous êtes en train de décrire.
- `Returns` vous permet de décrire le résultat de votre méthode.
- Dans la balise `Remarks`, vous pouvez marquer tout autre commentaire que vous jugerez utile.

Il ne vous reste plus qu’à remplir ces balises.

```
''' <summary>
''' Cette méthode effectue une addition entre deux entiers
''' </summary>
''' <param name="x">Premier membre de l'opération</param>
''' <param name="y">Deuxième membre de l'opération</param>
''' <returns>Le résultat de l'addition, qui est une
✂ opération entière</returns>
''' <remarks>Nous avons fait une opération entière car
''' une addition entre 2 entiers sera toujours
✂ entière</remarks>
```

```
    Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
        ' Retourne l'addition entre x et y
        ' Cette opération se fait sur nombres entiers
        Return x + y
    End Function
```

Maintenant, si vous compilez votre projet, vous trouverez dans le dossier de sortie un fichier XML contenant votre documentation.

```
<member name="M:TestAppliBackgroundWorker.Form1.Add(System
✂ .Int32,System.Int32)">
  <summary>
 Cette méthode effectue une addition entre deux entiers
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de l'addition, qui est une
 ✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une addition entre 2 entiers sera toujours entière<
 ✂ /remarks>
</member>
```

Maintenant, même si le format XML n'est pas le plus lisible du monde par un être humain, vous possédez quand même un fichier indépendant de votre programme qui permettra à une personne extérieure de comprendre ce que vous avez voulu faire.

Vous devez maintenant renseigner toutes les opérations :

```
    ''' <summary>
    ''' Cette méthode effectue une addition entre deux entiers
    ''' </summary>
    ''' <param name="x">Premier membre de l'opération</param>
    ''' <param name="y">Deuxième membre de l'opération<
✂ /param>
    ''' <returns>Le résultat de l'addition, qui est une
✂ opération entière</returns>
    ''' <remarks>Nous avons fait une opération entière car
    ''' une addition entre 2 entiers sera toujours
✂ entière</remarks>
    Public Function Add(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
        ' Retourne l'addition entre x et y
        ' Cette opération se fait sur nombres entiers
        Return x + y
    End Function
```

```
''' <summary>
''' Cette méthode effectue une soustraction entre deux
✂ entiers
''' </summary>
''' <param name="x">Premier membre de l'opération<
✂ /param>
''' <param name="y">Deuxième membre de l'opération<
✂ /param>
''' <returns>Le résultat de la soustraction, qui est
✂ une opération entière</returns>
''' <remarks>Nous avons fait une opération entière car
''' une soustraction entre 2 entiers sera toujours
✂ entière
''' Cependant, le résultat peut être négatif si x est
✂ plus grand que y</remarks>
Public Function Substract(ByVal x As Integer, ByVal y
✂ As Integer) As Integer
    ' Retourne la soustraction entre x et y
    ' Cette opération se fait sur nombres entiers
    Return x - y
End Function
''' <summary>
''' Cette méthode effectue une multiplication entre
✂ deux entiers
''' </summary>
''' <param name="x">Premier membre de l'opération<
✂ /param>
''' <param name="y">Deuxième membre de l'opération<
✂ /param>
''' <returns>Le résultat de la multiplication, qui est
✂ une opération entière</returns>
''' <remarks>Nous avons fait une opération entière car
''' une multiplication entre 2 entiers sera toujours
✂ entière
''' Cependant, le résultat peut être négatif si un des
✂ nombres est négatif</remarks>
Public Function Times(ByVal x As Integer, ByVal y As
✂ Integer) As Integer
    ' Retourne la multiplication entre x et y
    ' Cette opération se fait sur nombres entiers
    Return x * y
End Function
''' <summary>
''' Cette méthode effectue une division entre deux
✂ nombres décimaux
''' </summary>
''' <param name="x">Premier membre de l'opération<
✂ /param>
''' <param name="y">Deuxième membre de l'opération<
✂ /param>
```

```
    ''' <returns>Le résultat de la division, qui est une
   ✂ opération décimale</returns>
    ''' <remarks>Nous avons fait une opération décimale
   ✂ car
    ''' une division entre 2 nombres peut être décimale
    ''' De plus, le résultat peut être négatif si un des
   ✂ nombres est négatif</remarks>
    Public Function Divide(ByVal x As Double, ByVal y As
   ✂ Double) As Double
        ' Retourne la division entre x et y
        ' Cette opération se fait sur nombres décimaux
        ' En effet, le résultat d'une division n'étant pas
       ✂ exact,
        ' il est préférable de travailler avec des Double
        Return x / y
    End Function
```

Vous aurez maintenant un fichier de documentation XML complet à propos de votre projet.

```
<member name="M:TestAppliBackgroundWorker.Form1.Add(System
✂ .Int32,System.Int32)">
  <summary>
 Cette méthode effectue une addition entre deux entiers
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de l'addition, qui est une
 ✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une addition entre 2 entiers sera toujours entière<
 ✂ /remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .Substract(System.Int32,System.Int32)">
  <summary>
 Cette méthode effectue une soustraction entre deux
 ✂ entiers
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de la soustraction, qui est une
 ✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une soustraction entre 2 entiers sera toujours entière
 Cependant, le résultat peut être négatif si x est plus
 ✂ grand que y</remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .Times(System.Int32,System.Int32)">
  <summary>
 Cette méthode effectue une multiplication entre deux
 ✂ entiers
```

```
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de la multiplication, qui est une
 ✂ opération entière</returns>
  <remarks>Nous avons fait une opération entière car
  une multiplication entre 2 entiers sera toujours entière
 Cependant, le résultat peut être négatif si un des
 ✂ nombres est négatif</remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .Divide(System.Double,System.Double)">
  <summary>
 Cette méthode effectue une division entre deux nombres
 ✂ décimaux
 </summary>
  <param name="x">Premier membre de l'opération</param>
  <param name="y">Deuxième membre de l'opération</param>
  <returns>Le résultat de la division, qui est une
 ✂ opération décimale</returns>
  <remarks>Nous avons fait une opération décimale car
  une division entre 2 nombres peut être décimale
 De plus, le résultat peut être négatif si un des nombres
 ✂ est négatif</remarks>
</member><member name="M:TestAppliBackgroundWorker.Form1
✂ .BackgroundCount">
  <summary>
 Méthode de comptage utilisant un BackgroundWorker
 </summary>
  <remarks></remarks>
</member>
```

Bon, ce n'était déjà pas tellement lisible sur une méthode… Maintenant qu'il y en a quatre, ce n'est plus du tout possible de comprendre quoi que ce soit. Heureusement qu'il existe des outils capables d'analyser les fichiers de documentation XML pour en faire des fichiers plus lisibles pour un homme.

Le logiciel NDoc

Le logiciel de référence pour analyser les fichiers de documentation XML et les transformer en fichier d'aide est NDoc. À partir de votre DLL et du fichier de documentation XML qui a été généré, NDoc pourra vous fournir une documentation dans différents formats.

Déjà, préparons le projet.

1 Créez un projet de bibliothèque de classe que l'on appellera Calculatrice.

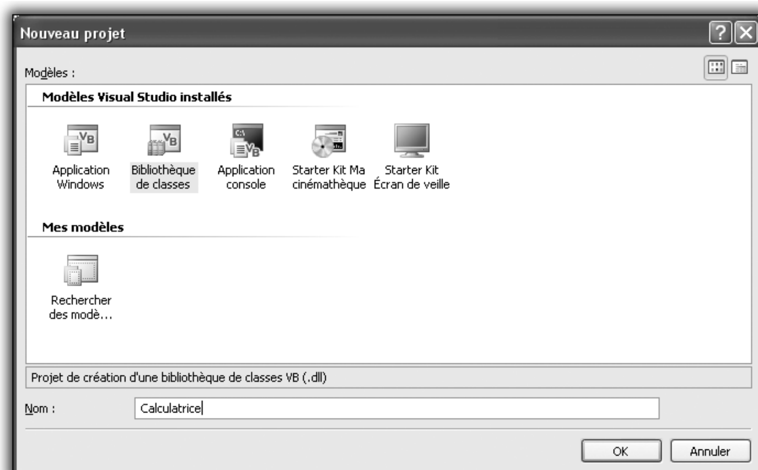


Figure 9.62 : *Nouvelle bibliothèque de classe*

2 Créez une nouvelle classe et mettez-y le code créé précédemment. Compilez. Vous devriez avoir plusieurs fichiers dans le répertoire de sortie.

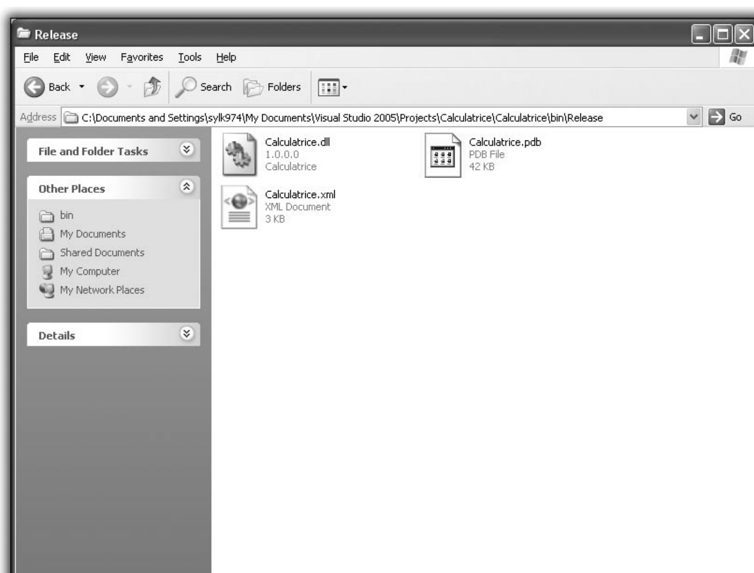


Figure 9.63 : *Fichiers générés par votre programme*

Laissons là un instant le code pour vous procurer NDoc. Le site officiel est http://ndoc.sourceforge.net/.

Vous pourrez y trouver de nombre d'aides et de détails à propos de cette application. Cependant, la version officielle de NDoc n'est pas compatible avec le framework .Net version 2.0. Fort heureusement, la communauté (NDoc est un projet OpenSource) a créé un portage pour la version 2 du framework .Net. En revanche, ce portage n'est pas officiel, et la version existante n'est pas stable. Mais elle permet quand même un nombre d'actions suffisant pour être intéressante. Vous pourrez l'obtenir sur le site http://www.kynosarges.de/NDoc.html.

NDoc nécessite une application de Microsoft : HTML Help Workshop. Il faudra vous la procurer avant de pouvoir utiliser NDoc. Elle est disponible à l'adresse http://www.microsoft.com/downloads/details.aspx?FamilyID=00535334-c8a6-452f-9aa0-d597d16580cc&displaylang=en.

Une fois NDoc téléchargé et décompressé, exécutez le fichier `Ndocgui.exe`.

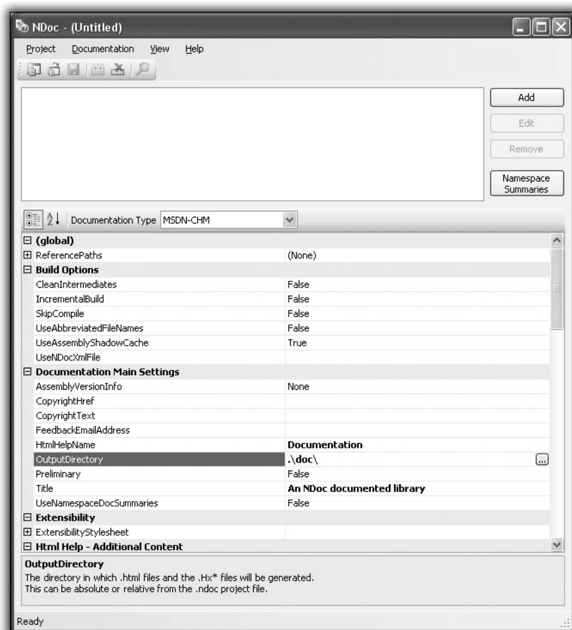


Figure 9.64 : *Lancement de NDoc*

De nombreuses options sont configurables dans NDoc. Il n'est pas nécessaire de s'y attarder pour l'instant. Cependant, dans la liste

déroulante, sélectionnez *MSDN-CHM*. Vous aurez de cette manière un fichier *CHM* proche de la documentation MSDN de Microsoft.

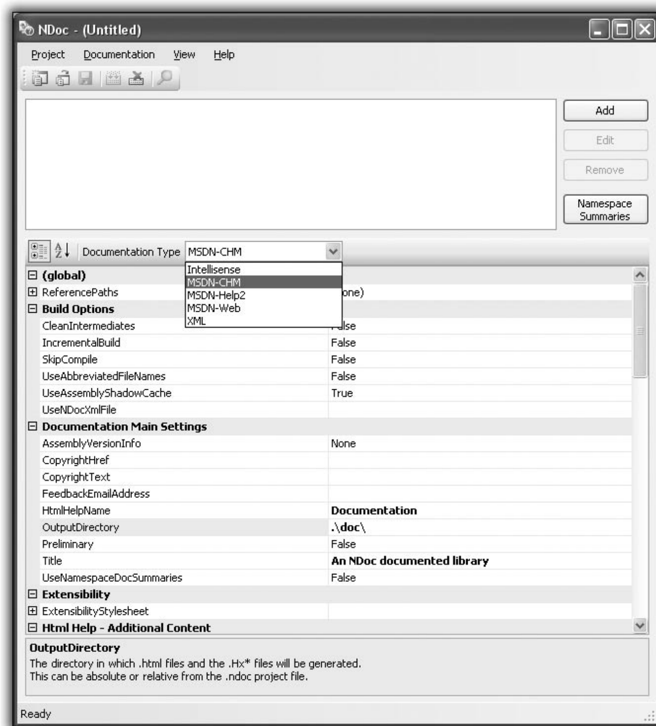


Figure 9.65 : *Sélection du type de documentation*

Maintenant, nous allons ajouter votre application pour que NDoc crée sa documentation. Pour cela, cliquez sur **Add**, et vous arriverez sur un menu où vous devrez sélectionner la DLL.

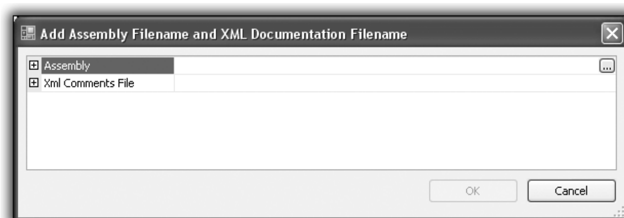


Figure 9.66 : *Sélection de la DLL*

En cliquant sur "**...**" vous pourrez choisir votre DLL *via* un explorateur de fichiers.



Figure 9.67 : *Exploration de fichiers*

Quand vous aurez sélectionné la DLL, NDoc cherchera automatiquement le fichier de documentation XML au même endroit. Faites OK pour valider l'ajout de votre DLL.

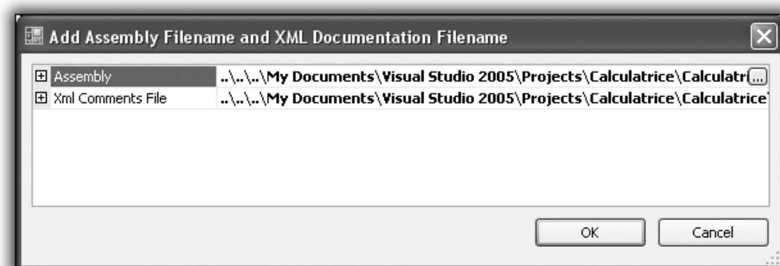


Figure 9.68 : *DLL + documentation XML*

Vous revenez maintenant sur l'écran principal de NDoc, avec votre DLL qui a été ajoutée à la liste des projets à documenter. Pour lancer la création de la documentation, il ne vous reste plus qu'à cliquer sur **Build docs**.

Figure 9.69 : Lancement de la création de la documentation

Vous pourrez voir le déroulement grâce à la *ProgressBar* de l'interface de NDoc, qui redeviendra normale une fois l'opération terminée.

Figure 9.70 : Progression du traitement

La documentation générée se trouve dans le dossier *doc* de NDoc. Elle s'appelle *Documentation.chm*.

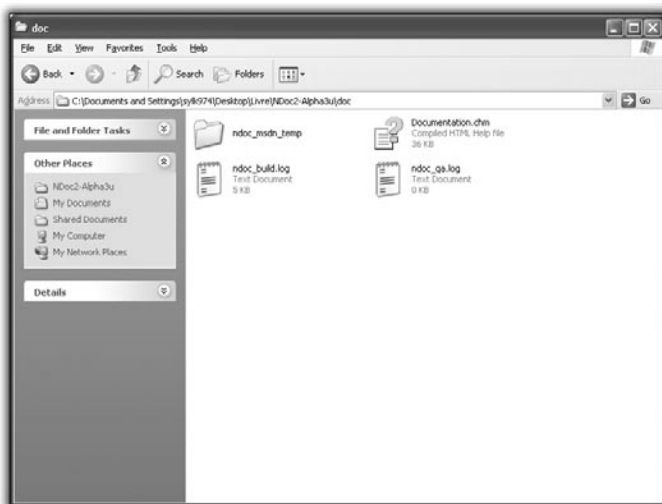


Figure 9.71 : *Fichier généré*

Maintenant, tout simplement en double-cliquant sur ce fichier, vous pouvez voir le résultat produit par NDoc.

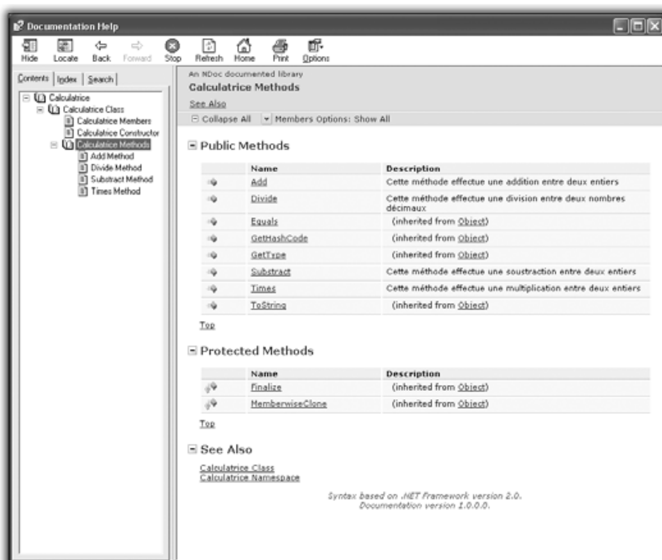


Figure 9.72 : *Documentation finale*

Reconnaissez qu'elle a un aspect très propre et très professionnel. Pourtant, vous conviendrez que cette étape n'a pas été particulièrement difficile. Elle demande un peu de méthode et de méthodologie, un bon outil et un peu de bonne volonté. Vos utilisateurs et vos collaborateurs vous en remercieront car, grâce à cela, ils n'auront pas besoin de décortiquer votre code pour savoir ce que vous avez voulu faire (aussi tentés qu'ils soient, ce qui n'est pas toujours le cas). Merci, NDoc !

9.8. Garder l'interactivité avec l'utilisateur

Après les derniers chapitres que vous avez lus, il est certain que vous avez passé un certain nombre d'étapes dans la connaissance de la programmation. La programmation objet, les modèles de conception sont autant de concepts que vous avez appréhendés et qui marquent déjà une évolution. Mais la route de la connaissance est encore longue, et vous allez maintenant vous heurter à un des concepts les plus subtils de la programmation : le multithreading.

Le multithreading est en fait la capacité de diviser vos programmes en plusieurs sous-programmes réalisant chacun un certain nombre d'opérations en parallèle.

Mais, avant d'entrer dans le vif du sujet, il faut que vous connaissiez quelques notions de base pour comprendre la suite de ce chapitre.

Tout d'abord, qu'est-ce qu'un processus ?



Processus

Un processus est un programme, c'est-à-dire un élément qui exécute du code machine.

Il y a autant de processus que de programmes, et chacun d'eux peut faire quelque chose de différent. Un peut faire du calcul, l'autre, de l'affichage, un autre peut même être un jeu complet.

1 Appuyez sur Ctrl+Alt+Supp, et allez dans le gestionnaire des tâches de Windows. Puis allez dans l'onglet **Processus**. Vous verrez la liste des processus qui sont en ce moment exécutés sur votre ordinateur.

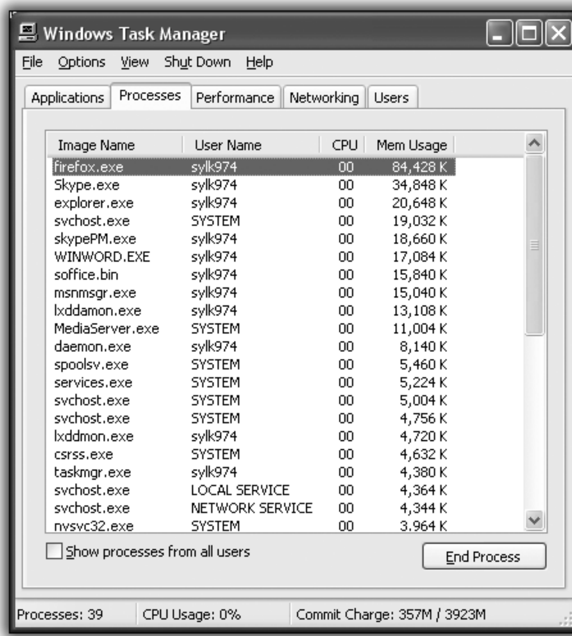


Figure 9.73 : *Processus exécuté sur votre ordinateur*

Comme vous pouvez le voir, énormément de programmes (et donc de processus) tournent en même temps sur votre ordinateur. Peut-être en connaissez-vous certains ?

Mais comment cela se peut-il ? On a vu qu'un programme était une liste d'instructions exécutées les unes après les autres. C'est effectivement le cas, sauf que votre système d'exploitation (ici Windows) leur fait exécuter leurs instructions à tour de rôle. Ils passent tous dans une file d'attente du système, puis chacun à son tour exécute un morceau de programme. C'est ensuite le tour d'un autre programme d'exécuter ses instructions, et ainsi de suite. Voici un schéma des états d'un processus qui sont chargés sur le système d'exploitation (voir Figure 9.74).

Le système d'exploitation va donc mettre un processus en sommeil, en activer un autre, le mettre en sommeil, en activer un autre, et ceci pour chacun des processus existants. Et, quand il a fini un tour complet, il recommence. Quel travail passionnant, me direz-vous...

Autre question qui se pose. Comment tous ces processus ne se marchent-ils pas sur les pieds ? Supposons qu'un de nos programmes modifie une de ses données. On vient de voir qu'il y avait un certain

nombre de processus qui tournaient en même temps sur notre machine. Or la mémoire de l'ordinateur est unique. Donc, rien n'empêche que tel programme ait une donnée commune avec tel autre.

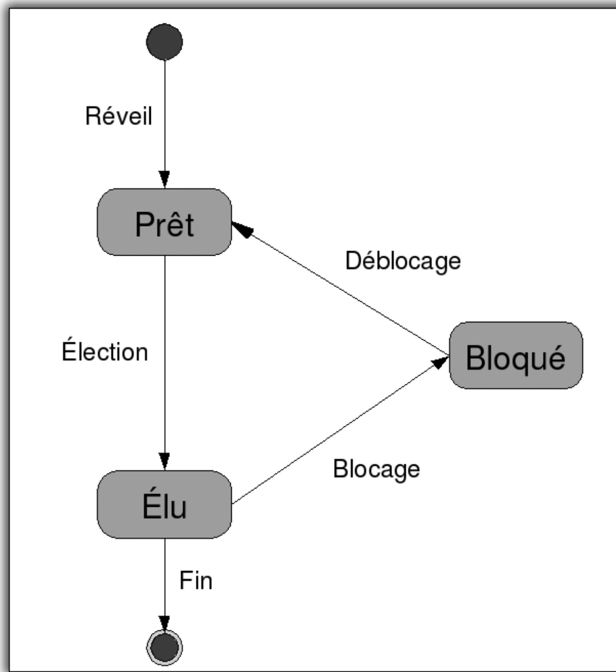


Figure 9.74 : *État d'un processus*

Là encore, c'est le rôle du système d'exploitation de gérer cela. Celui-ci va réserver un espace mémoire pour chacun des processus afin que celui-ci ait ses données propres et n'aille pas embêter les autres. Vous avez peut-être déjà eu l'occasion de voir une erreur qui disait `Access Violation`. Cela arrive quand un programme essaie d'accéder à une donnée ou à un espace mémoire qui ne lui appartient pas. Grâce à ce mécanisme, encore géré par le système d'exploitation (il en fait, des choses !), les modifications faites en mémoire par un programme ne se répercutent pas sur un autre. D'un autre côté, un programme ne pourra jamais voir directement les données d'un autre. De plus, étant donné que c'est le système d'exploitation qui décide de la quantité d'espace mémoire utilisée par un programme, vous pourriez avoir une erreur `OutOfMemoryException`, alors qu'il vous reste de l'espace en mémoire ou sur le disque dur.

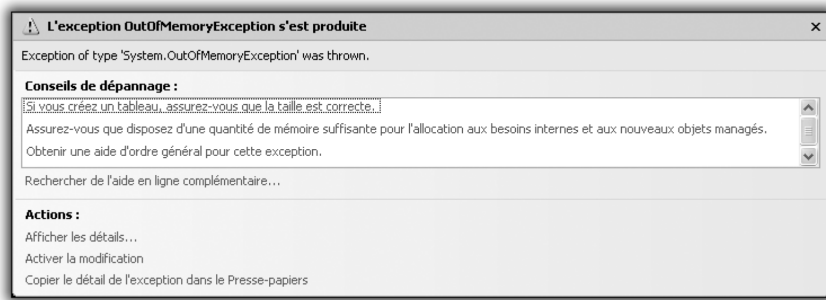


Figure 9.75 : *OutOfMemoryException : plus d'espace disponible pour votre processus*

On voit donc que le rôle du système d'exploitation est primordial pour la gestion de vos programmes.

Nous savons ce qu'est un processus ; maintenant, qu'est-ce qu'un thread ?



Thread

Un thread est également un processus, car il exécute des instructions. Cependant, il est lié à un processus principal dont il partage l'espace mémoire. On dit que les threads sont des processus légers.

Un thread est une sorte de sous-processus. Il vit à l'intérieur d'un processus et partage son espace mémoire, c'est-à-dire qu'il peut voir et agir sur les données de son processus père.

Au même titre qu'un processus, il est dans la file d'attente du système d'exploitation, qui le laissera à son tour exécuter un certain nombre d'instructions.

Maintenant que nous avons dégrossi les concepts de processus et de threads, nous allons voir ce qu'est le multithreading, quelle est son utilité, mais également quels problèmes cela peut poser. Nous verrons bien sûr comment répondre à ces problèmes. Comme je le disais en introduction de ce chapitre, le multithreading est un concept très très subtil, ainsi qu'une grosse source d'erreurs, même dans des applications professionnelles. À utiliser avec beaucoup d'attention !

Nous verrons ensuite un moyen simple de profiter du multithreading pour exécuter des instructions en tâche de fond, de manière à ne pas bloquer l'utilisateur.

Puis comment réagir sur l'interface graphique, de manière à pouvoir suivre ce qui est fait par vos threads. Et, enfin, comment gérer les erreurs dans les threads, car cela peut très vite devenir le bazar, et vous pouvez chercher des heures la source d'un problème qui est en fait dû à l'action d'un thread sur un autre qui a provoqué un état instable qui a fait planter votre programme. Du bonheur en perspective !

Introduction au multithreading et à ses problématiques

Maintenant que vous savez ce que sont les threads et les processus, vous vous demandez bien quel intérêt cela peut-il avoir. En ce qui concerne les processus, très peu. Il s'agit d'un programme, il exécute vos instructions, c'est très bien, mais vous ne pouvez pas faire grand-chose de plus avec.

Alors qu'un thread est un objet que vous pouvez contrôler dans vos programmes. Vous pouvez en instancier de nouveaux, les supprimer, les stopper, les démarrer, bref, les contrôler comme une donnée de vos programmes ! À ceci près qu'un thread est une donnée qui peut exécuter du code.

De ce fait, en instanciant plusieurs threads, vous pouvez exécuter du code en parallèle. On a vu que, comme le processus principal, un thread est un processus qui est dans la file d'attente des processus du système d'exploitation.

Imaginez qu'il n'y ait pas de thread supplémentaire dans votre processus. À chaque tour complet de la file d'attente du système d'exploitation, votre programme n'aura exécuté qu'un nombre d'instructions correspondant à une unité. Maintenant, si ce même processus utilise une centaine de threads, à chaque tour complet votre programme (c'est-à-dire le processus principal plus l'ensemble de ces threads fils) aura exécuté cent une unités d'instructions. Une pour chacun des cent threads plus une pour le processus principal.

Petit cas d'illustration

Rien de tel qu'une belle métaphore pour expliquer ce concept.

Imaginez que vous allez vous marier, et c'est le moment d'envoyer les invitations. Si vous êtes déjà marié, revenez quelques années en arrière à ce même moment. Et, si vous l'avez fait organiser par quelqu'un d'autre, imaginez que vous l'avez fait vous-même.

Comme vous avez énormément d'amis, il y a environ un millier de lettres à envoyer. Vous vous munissez des enveloppes, des adresses, des timbres, et avec beaucoup de bonne volonté vous vous y mettez. Au bout d'un certain temps (qui pour moi serait inenvisageable, je dois l'avouer, je n'ai pas beaucoup de patience), finalement, vous aurez réussi à envoyer toutes les lettres.

Maintenant, imaginez le même travail, sauf que vous avez un ami pour vous aider. Vous lui passez la moitié des enveloppes, la moitié des adresses, vous partagez les timbres, vous prenez bien soin de lui expliquer ce qu'il faut faire et vous vous y mettez à deux. Outre le fait que ce sera plus marrant et plus sympa, vous allez surtout mettre deux fois moins de temps.

Pareil, mais avec vingt amis. Vous mettrez vingt fois moins de temps. En revanche, si on essayait avec cinq cents, on pourrait penser que cela prendrait cinq cents fois moins de temps mais, finalement, vous vous rendez compte que le temps d'expliquer à chacune des cinq cents personnes ce qu'il faut faire, vous auriez pu envoyer au moins cinq mille lettres.

Il y a donc un nombre d'amis qui optimise le temps d'envoi de toutes vos lettres.

Le multithreading, c'est exactement cela.

- Les lettres à envoyer représentent le but de votre programme, ce qu'il doit faire.
- Les enveloppes, les adresses et les timbres représentent les données.
- Et vous et vos amis êtes des threads. (Ne vous inquiétez pas, on s'y fait bien !)

Mais vous avez vu qu'il a fallu expliquer à vos amis ce qu'il fallait faire. De la même manière, il vous faudra préparer les threads aux actions qu'ils devront effectuer.

De plus, vous avez dû partager les enveloppes et les timbres. Pareillement, vous aurez à vous arranger pour partager les données entre les threads.

En revanche, il n'est pas impossible qu'un de vos amis ait utilisé vos timbres parce qu'il n'en avait plus ou alors qu'il ait fini bien avant les autres. Cela peut arriver également au niveau des threads. Ils ne s'arrêteront pas en même temps et pourront être amenés à utiliser les mêmes données. Il s'agit là de la problématique principale du multithreading : l'accès concurrent aux données.

En effet, la principale source d'erreur lorsque l'on traite du multithreading est due à des données modifiées par des threads alors qu'un autre thread s'en servait.

Imaginez que vous vouliez enregistrer une émission à la télévision. Vous programmez l'enregistrement sur une chaîne donnée. Votre conjoint passe par là et change de chaîne pour regarder son émission. Et vous vous retrouvez avec le programme de cette chaîne plutôt que sur celui que vous vouliez enregistrer. Le problème a été posé par un accès concurrent à la télévision. Vous avez tous les deux voulu l'utiliser et le traitement de l'un a perturbé l'autre.

Maintenant que nous avons vu la théorie sur ce qu'est le multithreading, faisons une application pour illustrer concrètement ce qui se passe.

Application explicative

1. Tout d'abord, créons un nouveau projet (voir Figure 9.76).
2. On l'appellera `AppliTestMultiThread`. Ce sera une application Console simple pour présenter le concept de multithreading (voir Figure 9.77).

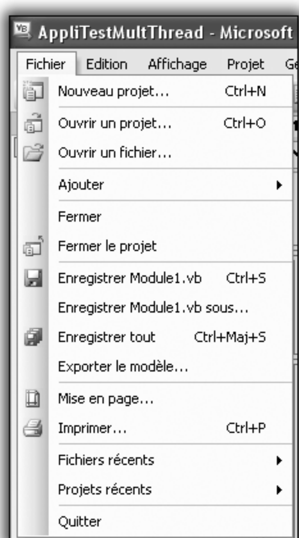


Figure 9.76 :
Création d'un nouveau projet

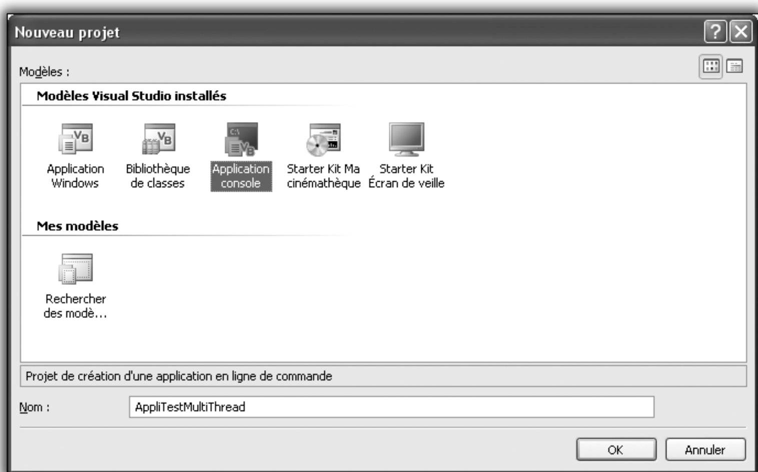


Figure 9.77 : *Nouvelle application Console.*

Dans un premier temps, notre application ne fera que boucler sur une valeur et l'écrire. Elle marquera ensuite la fin du programme, que nous finirons par un `ReadLine` pour que la fenêtre ne disparaisse pas tout de suite.

```
Sub Main()
   Dim count As Integer = 0
```

```
    For i = 1 To 20
       ’ Traitement à faire 100X
       ’count += 1
       Console.WriteLine("Dans le thread principal, i = "
                          & i.ToString)
    Next i

    Console.WriteLine("J’ai fini sans problème le thread
                       principal. Nombre de traitement = "
                       & count.ToString)
    Console.ReadLine()
End Sub
```

L’exécution de ce programme devrait vous ressortir ceci :

```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appli...
Dans le thread principal, i = 1
Dans le thread principal, i = 2
Dans le thread principal, i = 3
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread principal, i = 6
Dans le thread principal, i = 7
Dans le thread principal, i = 8
Dans le thread principal, i = 9
Dans le thread principal, i = 10
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread principal, i = 13
Dans le thread principal, i = 14
Dans le thread principal, i = 15
Dans le thread principal, i = 16
Dans le thread principal, i = 17
Dans le thread principal, i = 18
Dans le thread principal, i = 19
Dans le thread principal, i = 20
J’ai fini sans problème le thread principal. Nombre de traitement = 0
```

Figure 9.78 : *Exécution du programme*

Utilisons maintenant dans ce même programme un autre thread qui va faire la même chose. Pour cela, il faut déclarer une méthode qui sera le point d’entrée du thread, c’est-à-dire la première qu’il va exécuter. Dans notre cas, on aura une méthode qui fera la même boucle que celle du programme principal en précisant que c’est celle du thread secondaire.

```
Sub MakeLoop()
    Dim count As Integer = 0

    For i = 1 To 20
       count += 1
       Console.WriteLine("Dans le thread secondaire, i = "
                          & i.ToString)
    Next i
```

```
    Console.WriteLine("J'ai fini sans problème le thread
                      secondaire. Nombre de traitement = "
                      & count.ToString)
End Sub
```

Maintenant, pour utiliser un nouveau thread, il suffit tout simplement de l'instancier en précisant son point de départ, ici notre méthode `MakeLoop`. Il faut pour cela utiliser le mot-clé `AddressOf` suivi du nom de la méthode qui sera le point d'entrée. La classe de `Thread` est dans le namespace `Threading`. Ensuite, pour exécuter le thread il n'y a plus qu'à faire `Start`.

```
Imports System.Threading
Sub Main()
  Dim count As Integer = 0
  Dim threadSecondaire As New Thread(AddressOf MakeLoop)
  threadSecondaire.Start()

  Console.WriteLine("J'ai fini sans problème le thread
                    principal. Nombre de traitement = "
                    & count.ToString)
  Console.ReadLine()
End Sub
```

À l'exécution, nous obtiendrons ceci :

```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appli...
J'ai fini sans problème le thread principal. Nombre de traitement = 0
Dans le thread secondaire, i = 1
Dans le thread secondaire, i = 2
Dans le thread secondaire, i = 3
Dans le thread secondaire, i = 4
Dans le thread secondaire, i = 5
Dans le thread secondaire, i = 6
Dans le thread secondaire, i = 7
Dans le thread secondaire, i = 8
Dans le thread secondaire, i = 9
Dans le thread secondaire, i = 10
Dans le thread secondaire, i = 11
Dans le thread secondaire, i = 12
Dans le thread secondaire, i = 13
Dans le thread secondaire, i = 14
Dans le thread secondaire, i = 15
Dans le thread secondaire, i = 16
Dans le thread secondaire, i = 17
Dans le thread secondaire, i = 18
Dans le thread secondaire, i = 19
Dans le thread secondaire, i = 20
J'ai fini sans problème le thread secondaire. Nombre de traitement = 20
```

Figure 9.79 : *Exécution dans un autre thread*

Très bien, nous avons réussi à produire du code dans un autre thread. Mais nous remarquons une chose, c’est que la phrase de fin du thread principal a été écrite avant la première phrase du thread secondaire.

C’est tout à fait normal. Comme nous l’avons vu à la présentation du concept de thread, c’est un processus séparé. Le processus principal va donc continuer son exécution sans attendre le second thread. Celui-ci est bien exécuté en parallèle du premier.

Tout cela est très bien, mais ici on a juste déplacé le traitement. Il y a effectivement deux threads, mais il n’y en a qu’un seul qui travaille réellement. Remettons alors notre boucle d’origine dans le programme principal. Cependant, pour que l’exemple soit plus parlant, il va falloir ajouter une instruction dans les boucles, `Sleep`. Cette instruction permet de stopper le thread qui l’exécute pendant un instant donné, calculé en millisecondes. En faisant cela, on simule un traitement long, car en réalité faire une boucle de 20 est tellement rapide que l’on ne verra pas les subtilités du multithreading.

```
Sub Main()
   Dim count As Integer = 0
   Dim threadSecondaire As New Thread(AddressOf MakeLoop)
   threadSecondaire.Start()

   For count = 1 To 20
      'count += 1
      Thread.Sleep(0)
      Console.WriteLine("Dans le thread principal, i = "
                        & count.ToString)
   Next count

   Console.WriteLine("J'ai fini sans problème le thread
                     principal. Nombre de traitement = "
                     & count.ToString)
   Console.ReadLine()
End Sub
```



MakeLoop et Sleep

N’oubliez pas d’ajouter également l’instruction `Sleep` dans la méthode `MakeLoop`

Vous devriez obtenir un programme qui ressemble à ceci (la sortie peut varier d'un ordinateur à l'autre, car elle dépend de la vitesse de traitement de votre système d'exploitation) :

```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appli...
Dans le thread principal, i = 1
Dans le thread principal, i = 2
Dans le thread principal, i = 3
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread principal, i = 6
Dans le thread principal, i = 7
Dans le thread principal, i = 8
Dans le thread secondaire, i = 1
Dans le thread principal, i = 9
Dans le thread secondaire, i = 2
Dans le thread principal, i = 10
Dans le thread secondaire, i = 3
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread secondaire, i = 4
Dans le thread principal, i = 13
Dans le thread secondaire, i = 5
Dans le thread principal, i = 14
Dans le thread secondaire, i = 6
Dans le thread principal, i = 15
Dans le thread secondaire, i = 7
Dans le thread principal, i = 16
Dans le thread secondaire, i = 8
Dans le thread principal, i = 17
Dans le thread secondaire, i = 9
Dans le thread principal, i = 18
Dans le thread principal, i = 19
Dans le thread principal, i = 20
J'ai fini sans problème le thread principal. Nombre de traitement = 21
Dans le thread secondaire, i = 10
Dans le thread secondaire, i = 11
Dans le thread secondaire, i = 12
Dans le thread secondaire, i = 13
Dans le thread secondaire, i = 14
Dans le thread secondaire, i = 15
Dans le thread secondaire, i = 16
Dans le thread secondaire, i = 17
Dans le thread secondaire, i = 18
Dans le thread secondaire, i = 19
Dans le thread secondaire, i = 20
J'ai fini sans problème le thread secondaire. Nombre de traitement = 20
_
```

Figure 9.80 : *Votre première application multithread*

Comme vous pouvez le constater, l'exécution dans l'un ou l'autre des deux threads est un peu aléatoire. Tantôt c'est le premier qui fait huit tours de boucle. Ensuite, ils alternent chacun leur tour. Puis le second thread finit, et donc le premier termine à son tour.

Jouez maintenant à modifier les valeurs de `Sleep` et le nombre de tours de boucle, vous verrez qu'utiliser deux threads est plus rapide que n'en utiliser qu'un seul pour le même traitement.

Tout se passe ici pour le mieux, on ne voit aucun problème, et pourtant nous avons optimisé notre temps de traitement, sans remettre en cause la pérennité de notre programme. Facile, il n'y a aucune donnée commune

utilisée par nos deux threads. Qu'à cela ne tienne, `i`, notre variable de boucle, va devenir une donnée globale qui sera commune à nos deux threads. (En vérité, ceci n'est pas recommandé, mais c'est pour vous faire toucher concrètement le problème d'accès concurrent aux données.)

```
Module Module1
   Dim i As Integer

'réécrire le code de MakeLoop également

Sub Main()
  Dim count As Integer = 0
  Dim threadSecondaire As New Thread(AddressOf MakeLoop)
  threadSecondaire.Start()

  For i = 1 To 20
      count += 1
      Thread.Sleep(0)
      Console.WriteLine("Dans le thread principal, i = "
                         & i.ToString)
  Next count

  Console.WriteLine("J'ai fini sans problème le thread
                    principal. Nombre de traitement = "
                    & count.ToString)
  Console.ReadLine()
Endd Sub

End Module
```

Et voici notre sortie : (voir Figure 9.81)

Et là, catastrophe... chacun de nos threads a fait la moitié de son travail. Au lieu d'avoir fait vingt opérations chacun, le premier en a fait treize, le second, huit, pas du tout ce qui était prévu à l'origine.

C'est là un problème d'accès concurrent aux données, mais nous allons voir que l'on peut y remédier.

```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appli...
Dans le thread principal, i = 1
Dans le thread secondaire, i = 1
Dans le thread principal, i = 2
Dans le thread secondaire, i = 3
Dans le thread principal, i = 3
Dans le thread secondaire, i = 5
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread secondaire, i = 7
Dans le thread principal, i = 6
Dans le thread secondaire, i = 10
Dans le thread principal, i = 7
Dans le thread secondaire, i = 12
Dans le thread principal, i = 8
Dans le thread secondaire, i = 14
Dans le thread principal, i = 9
Dans le thread principal, i = 10
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread principal, i = 13
J'ai fini sans problème le thread principal. Nombre de traitement = 13
Dans le thread secondaire, i = 16
J'ai fini sans problème le thread secondaire. Nombre de traitement = 8
```

Figure 9.81 : *Accès concurrent aux données*

Une solution naïve mais efficace : l'exclusion mutuelle

Dans notre programme précédent, le problème était dû à l'utilisation de la variable de boucle par deux threads différents. Donc, lorsqu'un des deux threads faisait un tour de boucle, il incrémentait cette variable et l'autre thread sautait un tour.

Comment régler ce problème ? La meilleure réponse possible : s'arranger pour qu'ils n'utilisent pas la même donnée. Mais comme il arrivera des cas où ce n'est pas possible (alors qu'ici on pourrait), nous allons chercher autre chose.

Comme j'aime bien les analogies, je vais tenter de vous faire toucher du doigt la solution par un cas de tous les jours : vous êtes au vidéoclub, vous voyez un film que vous voulez. Manque de chance, quelqu'un le prend avant vous. Que faites-vous ? Malheureusement, vous prenez votre mal en patience, et vous attendez que l'autre client le rapporte. L'analogie avec notre programme :

- Vous et l'autre client êtes des threads (vous vous y faites mieux la deuxième fois ?).
- La cassette représente la donnée partagée.
- Et notre solution, c'est l'attente. Cette solution se nomme l'exclusion mutuelle. Car, lorsqu'un thread accède à une donnée, il en bloque l'accès aux autres threads. De cette manière, il assure son traitement et rend ensuite la main.

Pour utiliser cette solution, il nous faut un verrou. Ce sera en fait un objet dans notre programme dont le seul but sera de dire aux threads *Attention c'est verrouillé, personne ne passe*. Une instance d'`Object` fera très bien l'affaire. Nous sommes obligés d'utiliser un objet supplémentaire car on ne peut pas utiliser comme verrou des objets qui sont amenés à changer.

```
Dim locker As Object = New Object()
```

Maintenant que nous avons un verrou, encore faut-il verrouiller la donnée quand on l'utilise. Ceci est fait avec le mot-clé `SynLock`, qui a comme argument l'objet verrou que l'on va utiliser. `SynLock` définit un bloc d'instructions qui seront toutes exécutées sans qu'aucune modification n'ait pu être faite sur le verrou.

```
Dim locker As Object = New Object()

Sub MakeLoop()
  Dim count As Integer = 0

  SyncLock locker

    For i = 1 To 20
      count += 1
      Thread.Sleep(0)
      Console.WriteLine("Dans le thread secondaire, i = "
                       & i.ToString)
    Next i

  End SyncLock

  Console.WriteLine("J'ai fini sans problème le thread
                    secondaire. Nombre de traitement = "
                    & count.ToString)
End Sub

Sub Main()
  Dim count As Integer = 0
```

```
  Dim threadSecondaire As New Thread(AddressOf MakeLoop)
  threadSecondaire.Start()

    SyncLock locker

      For i = 1 To 20
        count += 1
        Thread.Sleep(0)
        Console.WriteLine("Dans le thread principal, i = "
                          & count.ToString)
      Next i

    End SyncLock

    Console.WriteLine("J'ai fini sans problème le thread principal.
                    Nombre de traitement = " & count.ToString)
    Console.ReadLine()
End Sub
```

Voici donc votre programme protégé des cas d'accès concurrent aux données. Le thread principal s'exécute. Il exécute le thread secondaire et continue son exécution. Il verrouille la donnée et entame sa boucle. De son côté, le thread secondaire, après s'être initialisé, tente de verrouiller la donnée. Or celle-ci a déjà été verrouillée par le thread principal, donc le thread secondaire attend. Le thread principal finit sa boucle et libère donc le verrou. Il finit ensuite son traitement. Le thread secondaire, qui attendait le verrou, voit celui-ci libéré. Il peut alors entamer l'exécution de sa boucle. C'est lui maintenant qui a mis le verrou. Il finit sa boucle. Fin du programme. Vous avez tout suivi ? Sinon voici l'illustration : (voir Figure 9.82)

En revanche, le revers de la médaille réside dans l'attente de libération du verrou. En effet, nous avons bien vu que, pendant le verrou, l'un des threads devait attendre l'autre. La performance est le prix de la sécurité.

De plus, dans de plus gros programmes, vous pouvez être amené à utiliser plusieurs verrous. Un problème bien connu dans l'utilisation des verrous est celui des verrous mortels : un thread 1 verrouille une donnée avec un verrou 1. Un autre thread 2 verrouille une autre donnée avec un verrou 2. Mais le thread 1 veut maintenant utiliser une donnée verrouillée par le verrou 2. Il attend donc. Et, de son côté, le thread 2 veut exécuter une instruction sur une donnée utilisée par le thread 1,

donc verrouillée par le verrou 1. Thread 1 attend Verrou 2 pour débloquer Verrou 1, mais Thread 2 attend Verrou 1 pour débloquer Verrou 2.

```
file:///C:/Documents and Settings/sylk974/My Documents/Visual Studio 2005/Projects/Appli...
Dans le thread principal, i = 1
Dans le thread principal, i = 2
Dans le thread principal, i = 3
Dans le thread principal, i = 4
Dans le thread principal, i = 5
Dans le thread principal, i = 6
Dans le thread principal, i = 7
Dans le thread principal, i = 8
Dans le thread principal, i = 9
Dans le thread principal, i = 10
Dans le thread principal, i = 11
Dans le thread principal, i = 12
Dans le thread principal, i = 13
Dans le thread principal, i = 14
Dans le thread principal, i = 15
Dans le thread principal, i = 16
Dans le thread principal, i = 17
Dans le thread principal, i = 18
Dans le thread principal, i = 19
Dans le thread principal, i = 20
J'ai fini sans problème le thread principal. Nombre de traitement = 20
Dans le thread secondaire, i = 1
Dans le thread secondaire, i = 2
Dans le thread secondaire, i = 3
Dans le thread secondaire, i = 4
Dans le thread secondaire, i = 5
Dans le thread secondaire, i = 6
Dans le thread secondaire, i = 7
Dans le thread secondaire, i = 8
Dans le thread secondaire, i = 9
Dans le thread secondaire, i = 10
Dans le thread secondaire, i = 11
Dans le thread secondaire, i = 12
Dans le thread secondaire, i = 13
Dans le thread secondaire, i = 14
Dans le thread secondaire, i = 15
Dans le thread secondaire, i = 16
Dans le thread secondaire, i = 17
Dans le thread secondaire, i = 18
Dans le thread secondaire, i = 19
Dans le thread secondaire, i = 20
J'ai fini sans problème le thread secondaire. Nombre de traitement = 20
```

Figure 9.82 : *Solution d'exclusion mutuelle*

Ils ne pourront jamais se débloquer car l'un attend l'autre qui attend l'un (comme l'œuf et la poule) : ceci est un cas de verrou mortel, ou `DeadLock`.

Vous connaissez maintenant les principales possibilités du multithreading, ainsi que ses problématiques principales, mais aussi une solution pour y remédier. N'hésitez pas à pratiquer et à tester les moindres cas particuliers, car le multithreading est une notion bien subtile. Même des programmeurs professionnels font beaucoup d'erreurs lorsqu'ils introduisent des threads dans leurs applications. De plus, la nature aléatoire de l'exécution rend le débogage très difficile. La seule vraie technique pour maîtriser les threads, la curiosité et la

pratique : étudier la théorie pour comprendre les principes et les coder pour voir la réalité des choses. Bon courage !

Le principe des tâches de fond

Contrôler des threads à la main offre bien des possibilités, mais on a également vu que cela pouvait poser pas mal de problèmes et obligeait à être particulièrement vigilant.

Heureusement, le framework .Net offre un mécanisme qui permet de bénéficier d'une capacité inhérente aux threads, l'exécution en tâche de fond. Vous pourrez de cette manière exécuter du code en arrière-plan et remonter des événements graphiques pour garder de l'interactivité dans votre programme.

1 Tout d'abord, créons un nouveau programme que nous appellerons `TestAppliBackgroundWorker`. Cette fois-ci, nous ferons une application graphique.

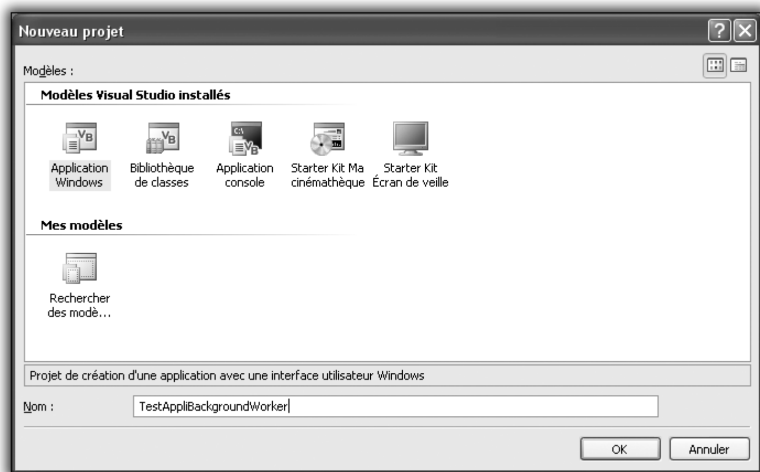


Figure 9.83 : *Application de tâche de fond*

2 Créez une interface graphique contenant au moins une *ProgressBar* et de quoi définir un nombre et lancer une action. Voici un modèle dont vous pourriez vous inspirer.

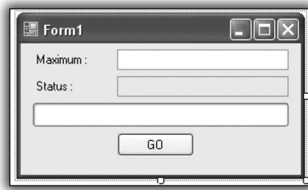


Figure 9.84 :
Interface graphique

Créez maintenant une méthode qui va compter. Vous pourrez définir la limite haute en utilisant votre interface et vous indiquerez l'état d'avancement dans la `ProgressBar`.

```
Public Sub Count()
    Dim count As Integer = Integer.Parse(TextBox1.Text)
    ProgressBar1.Maximum = count

    TextBox2.Text = "Launching count..."

    For i As Integer = 1 To count
        ProgressBar1.Value = i
        TextBox2.Text = "" & i.ToString()
    Next i

    TextBox2.Text = "Count finished."
End Sub
```

Maintenant, liez cette méthode à votre bouton, exécutez votre programme. Avec une petite limite haute, pas de souci :

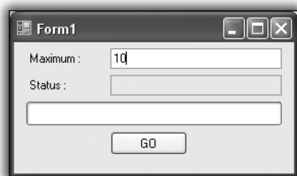


Figure 9.85 :
Programme avec dix itérations

Le programme va très vite, mais on voit la `ProgressBar` défiler, le comptage se faire, et enfin on voit le message final.

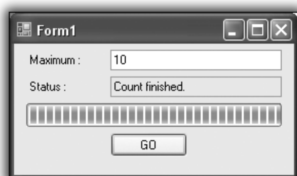


Figure 9.86 :
Succès avec dix itérations

Jusqu'ici tout va bien. Que se passe-t-il avec un nombre beaucoup plus gros, comme 100 000 ou 1 million :

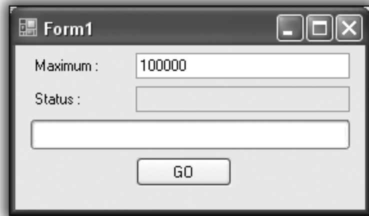


Figure 9.87 :
Programme avec un grand nombre

Nous voyons la barre avancer un peu puis, au bout d'un moment, vous remarquerez qu'elle n'avance plus. Si à ce moment-là on essaie de cliquer un peu partout pour voir ce qui se passe, pas de réaction, mais un message *Not Responding* sur la fenêtre.

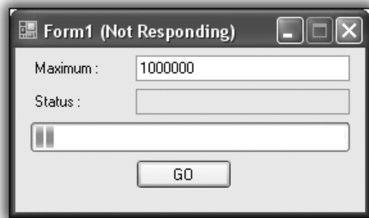


Figure 9.88 :
Blocage de l'application

Quelle explication à cela ? Il se trouve que, si votre programme partage le même thread que votre élément graphique, tout traitement de votre programme, s'il est trop long, peut bloquer l'affichage de cet élément graphique. Vous reprendrez la main seulement quand il aura fini son traitement.

Dans notre cas, c'est le même thread qui a créé les éléments de l'interface graphique (le formulaire, le bouton, les labels...) et qui fait le traitement. En effet, nous n'avons pas encore inclus dans notre programme un quelconque mécanisme de multithreading. Lorsque nous avons testé avec 10 comme valeur, pas de souci car le traitement était assez rapide pour qu'on ne s'aperçoive pas du blocage de l'interface graphique. Mais, dès lors qu'on a mis une limite maximale plus grande, nous avons senti ce blocage. Nous avons pu voir le début de l'avancement, car le traitement n'était pas considéré comme assez grand pour bloquer l'affichage de l'interface graphique, mais au moment où il a dépassé cette limite il nous est devenu impossible de récupérer la

main. Ce sera possible en fin de traitement mais il est impossible d'en voir le déroulement.

.Net propose un mécanisme un peu plus évolué qu'un thread, mais tout aussi simple à utiliser, qui nous permet d'exécuter du code en parallèle, de notifier de la progression et de finaliser l'exécution de celui-ci : le `BackgroundWorker`.

Pour ajouter un *BackgroundWorker* dans votre programme, il vous suffit de l'ajouter depuis la liste des contrôles vers votre interface.

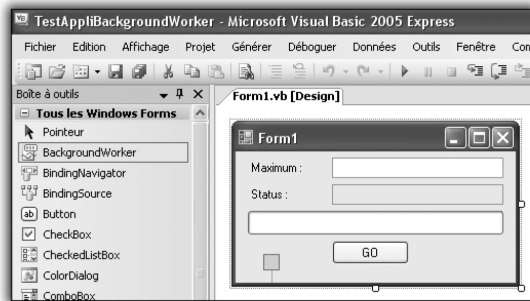


Figure 9.89 : *Ajout d'un BackgroundWorker au programme*

Il apparaît maintenant dans votre programme, mais en dehors de votre interface graphique.

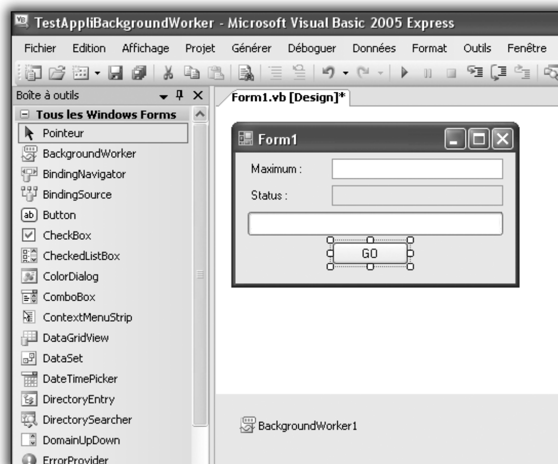


Figure 9.90 : *Intégration du BackgroundWorker*

Le `BackgroundWorker` se manie comme un objet normal, avec des propriétés, des attributs et des méthodes. La première chose à faire avec

le BackgroundWorker est de l'autoriser à notifier sa progression. Sans cette étape, il ne vous sera pas possible d'utiliser la méthode ReportProgress, qui permet de donner l'état d'avancement du BackgroundWorker.

```
BackgroundWorker1.WorkerReportsProgress = True
```

La première chose à définir est l'action à effectuer. Pour faire ceci, il faut traiter l'événement DoWork du BackgroundWorker. Il suffit d'aller dans l'écran des Propriétés et, dans la partie événement, de cliquer sur la partie DoWork.

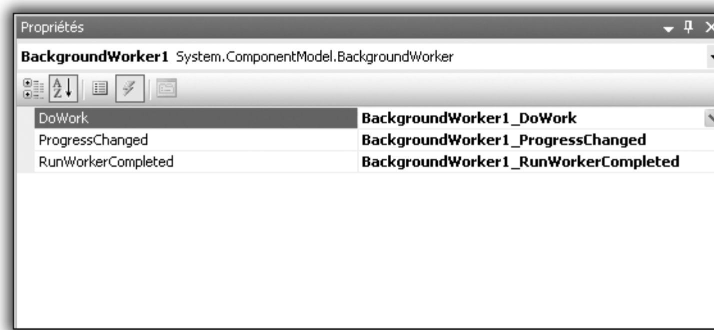


Figure 9.91 : *Définition de l'action principale*

En double-cliquant sur cet événement, Visual Studio vous crée le squelette de la méthode qui va faire l'action principale. Dans notre programme, l'action principale est de boucler sur une limite haute que vous aurez définie avant :

```
Public Sub BackgroundWorker1_DoWork(ByVal sender As
✂ System.Object, ByVal e As System.ComponentModel
✂ .DoWorkEventArgs) Handles BackgroundWorker1.DoWork
    Dim i As Integer
    For i = 0 To max
        counter = i
        BackgroundWorker1.ReportProgress(i * 100 /
      ✂ max)
    Next
End Sub
```

La limite haute, max, devra avoir été définie préalablement. Ceci se fera lors du Click sur le bouton. Nous verrons un peu plus loin le code de l'appel qui contient la création et le lancement du BackgroundWorker.

Nous remarquons une instruction dont je n'ai pas parlé : ReportProgress. Cette instruction correspond à la notification de l'état d'avancement du BackgroundWorker. Cet avancement correspond à un pourcentage. Grâce à un produit en croix avec la limite haute, on obtient le pourcentage d'avancement. Cette méthode ReportProgress lance l'événement ProgressChange. Il vous faut maintenant définir comment va être traité cet événement. Pour cela, de la même manière que pour DoWork, double-cliquez sur l'événement ProgressChange dans la fenêtre de Propriétés. Visual Studio va créer le squelette de la méthode dans laquelle vous pourrez mettre le traitement. À chaque nouvelle étape d'avancement, nous marquerons l'état d'avancement en cours, et nous ferons avancer la ProgressBar.

```
 Public Sub BackgroundWorker1_ProgressChanged(ByVal
✂ sender As System.Object, ByVal e As System
✂ .ComponentModel.ProgressChangedEventArgs) Handles
✂ BackgroundWorker1.ProgressChanged
     ProgressBar1.Value = e.ProgressPercentage
     TextBox2.Text = "" & counter.ToString()
 End Sub
```

e.ProgressPercentage correspond au nombre qui aura été passé en argument dans la méthode ReportProgress. C'est comme ceci que vous le retrouverez, ce qui permet de faire le lien entre le traitement principal de votre programme et la notification de l'avancement de celui-ci.

Enfin, il vous reste à définir l'action à faire en fin de traitement. Une fois encore, un événement correspond à cela : RunWorkerCompleted. Comme pour DoWork et ProgressChange, double-cliquez sur l'événement correspondant dans la fenêtre de Propriétés et Visual Studio vous créera le squelette de la méthode. On y marquera simplement la fin de traitement dans le Label de votre programme.

```
 Public Sub BackgroundWorker1_RunWorkerCompleted(ByVal
✂ sender As System.Object, ByVal e As System
✂ .ComponentModel.RunWorkerCompletedEventArgs) Handles
✂ BackgroundWorker1.RunWorkerCompleted
     TextBox2.Text = "Background count finished."
 End Sub
```

Maintenant que l'ensemble des événements du BackgroundWorker est défini, il ne reste plus qu'à préparer son lancement et à l'affecter au Click du bouton. Cliquez sur le bouton, et double-cliquez sur l'événement Click dans la fenêtre de Propriétés. Dans cette partie, on créera la limite haute pour le comptage, et on lancera le traitement du

BackgroundWorker après avoir prévenu du lancement du traitement. Voici ce que nous obtiendrons :

```
Public Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
    max = Integer.Parse(TextBox1.Text)
    ProgressBar1.Maximum = 100
    TextBox2.Text = "Launching background count..."
    BackgroundWorker1.RunWorkerAsync()
End Sub
```



Accessibilité

Il faudra avoir défini "max" dans une partie du programme accessible par toutes les méthodes.

Réessayons maintenant le lancement du programme avec un grand nombre.

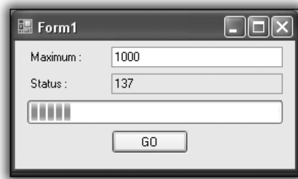


Figure 9.92 :
Utilisation du BackgroundWorker

Nous voyons que le programme s'exécute sans bloquer, et nous pouvons suivre l'avancement de notre boucle, jusqu'à sa fin.

Le BackgroundWorker est réellement un objet simple d'utilisation qui permet de profiter des bienfaits du multithreading pour exécuter une tâche en arrière-plan. Si une partie de votre programme doit prendre un peu de temps, il est conseillé d'utiliser cet objet plutôt qu'essayer de le gérer à travers des threads, surtout si vous devez agir sur une interface graphique. En effet, avec le framework .Net, la gestion des interfaces graphiques est un peu particulière dans un contexte multithreading. C'est ce que nous allons voir dans la partie qui suit.

Comment agir sur l'interface utilisateur ?

Merci au BackgroundWorker, nous avons pu dans la partie précédente effectuer une action en tâche de fond, notifier notre progression à

l'utilisateur, faire un traitement de fin, et tout cela sans bloquer l'affichage et sans perdre d'interactivité sur notre application.

Supposons maintenant que ce `BackgroundWorker` n'existe pas, et essayons d'utiliser les autres connaissances apprises dans le chapitre.

Vous vous en doutez bien, nous allons utiliser des threads pour effectuer le même traitement.

Utilisation des threads

Nous allons utiliser la même application, mais nous allons supprimer tout le code relatif au `BackgroundWorker`. Il ne reste plus grand-chose, me direz-vous, c'est pourquoi nous allons remplacer ce code par une gestion de threads créée par nos soins.



Reportez-vous à ce sujet au début de ce chapitre.

Comme nous l'avons vu, il faut créer à notre thread un point d'entrée, une méthode qui marquera le début de son action. Celle-ci va effectuer notre action de comptage et retransmettre à l'interface graphique les différentes actions effectuées. Nous l'appellerons `count` :

```
Public Sub Count()
    Dim count As Integer = Integer.Parse(TextBox1
  ✂ .Text)
    Dim i As Integer
    ProgressBar1.Maximum = count


    TextBox2.Text = "Launching count..."

    For i As Integer = 1 To count
        ProgressBar1.Value = i
        TextBox2.Text = "" & i.ToString()
    Next i

    TextBox2.Text = "Count finished."
End Sub
```

Très bien, maintenant que notre méthode de traitement est créée, il nous faut instancier un thread et lui spécifier que son point d'entrée sera la méthode `count` :

```
Dim thread1 As New Threading.Thread(AddressOf Count)
  thread1.Name = "ThreadCount"
  thread1.Start()
```

Parfait, il n'y a désormais plus qu'à lier cette action au `Click` du bouton de notre interface :

```
Public Sub Button1_Click(ByVal sender As System.Object,
   ByVal e As System.EventArgs) Handles Button1.Click
   Dim thread1 As New Threading.Thread(AddressOf Count)
   thread1.Name = "ThreadCount"
   thread1.Start()
End Sub
```

Notre transformation est maintenant prête, notre application va effectuer exactement la même tâche qu'avec le `BackgroundWorker`, mais en utilisant directement un thread. On exécute :

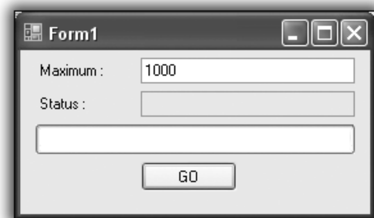


Figure 9.93 :
Exécution utilisant un thread secondaire

Et là, contre tout attente :

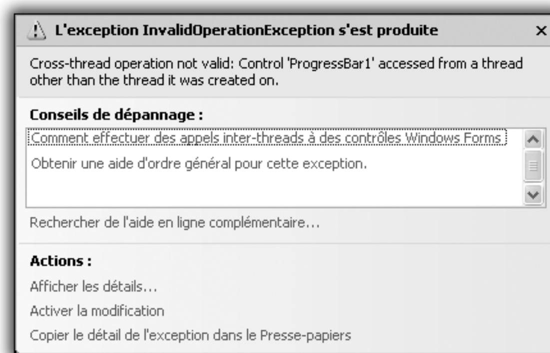


Figure 9.94 :
Crash de l'application

Notre application se termine avec une erreur *Cross Thread Exception*.

L'exception "Cross thread"

Voici la particularité dont je vous parlais concernant .Net et la gestion des interfaces graphiques dans un contexte multithread.

Le framework .Net met en place une protection qui empêche un thread de modifier une interface graphique si celle-ci n'a pas été construite dans le thread qui appelle la modification.



> **Éléments d'interface graphique**
> Par élément d'interface graphique, on entend tout objet dérivant de la classe `Control`, tels les `Form`, `Label`, `Button`, `ProgressBar`, `TextBox`…

Dans notre application, le processus principal crée la fenêtre.

D'un autre côté, nous créons pour faire notre traitement un autre thread qui exécutera la méthode `Count`. Or, dans cette méthode, nous essayons de modifier la `ProgressBar` :

```
ProgressBar1.Maximum = count
```

Cependant, nous avons vu qu'elle avait été créée dans le processus principal. C'est lors de cet accès que le framework se protège et envoie cette fameuse erreur *Cross thread exception*.

À ce point-là, nous sommes un peu coincés… On ne sait pas dire "*Thread 1, fais cette instruction, puis Thread 2, fais cette instruction, et revenons à Thread 2*". Car ce serait la solution. Nous effectuons le traitement non graphique dans notre thread secondaire, puis au moment de traiter les éléments graphiques on repasse dans le thread d'origine.

C'est à peu près la solution, sauf qu'on ne peut pas traiter le passage interthread par instructions. Il faudra le faire à partir de méthodes, en utilisant des délégués.

La solution : les délégués et la méthode Invoke

La solution est de faire exécuter par un `Control` toute méthode qui le modifie dans son thread d'origine.

Par exemple, nous avons vu que `count` modifie la `ProgressBar`. Donc, si le thread qui appelle `count` n'est pas celui qui a créé la `ProgressBar`, il faudra faire l'échange. Le thread qui a créé la `ProgressBar` va alors exécuter `count`.

La méthode qui permet de faire cela est la méthode `Invoke`. C'est une méthode qui existe pour tous les objets de type `Control` et qui va renvoyer l'exécution d'une autre méthode au thread d'origine.

Pour pouvoir utiliser `Invoke`, il faut en revanche créer un délégué correspondant à la méthode à *Invoker*.



Reportez-vous à ce sujet au chapitre ***Dialoguez avec un ordinateur****.*

```
Public Delegate Sub DelegateCounter()
```

Nous avons créé le délégué qui correspond à notre méthode `count`.

Maintenant, comme nous savons que notre méthode `count` va modifier la `ProgressBar`, au moment d'appeler `count` il nous faudra, plutôt que le faire directement, utiliser `Invoke`. `Invoke` prend comme paramètre la méthode à *Invoker*.

```
Dim myDelegate As New DelegateCounter(AddressOf Count)
ProgressBar1.Invoke(myDelegate)
```

De cette manière, l'accès au `Control` est sécurisé. Vous pouvez maintenant exécuter de nouveau l'application, qui pourra se poursuivre jusqu'à la fin.



Count

La méthode `count` modifie également d'autres `Control`, comme la `TextBox` ou le `Label`. Or ces éléments ont été créés dans le même thread que la `ProgressBar`, donc l'`Invoke` de celle-ci fonctionne également pour eux.

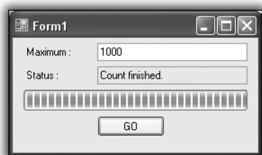


Figure 9.95 :
Exécution utilisant l'Invoke

Une bonne pratique concernant l'Invoke est l'utilisation d'une méthode Invoker qui va faire la vérification de la nécessité de l'Invoke. En effet, les objets de type Control possèdent également une propriété InvokeRequired qui permet de savoir à l'exécution s'il est nécessaire de faire un Invoke ou non.

De cette manière si l'Invoke est nécessaire, vous le faites, sinon vous appelez la méthode normalement. Il est recommandé de faire ceci car faire un Invoke est gourmand en ressources et peut réduire vos performances. L'utilisation de la propriété InvokeRequired permet d'éviter de le faire systématiquement. En définissant cette vérification dans une autre méthode, vous optimisez vos appels, et il suffit ensuite d'appeler l'Invoker pour garder la sécurité tout en limitant les pertes de performances.

```
Public Sub Invoker()
    Dim myDelegate As New DelegateCounter(AddressOf
   ✂ Count)
    If (ProgressBar1.InvokeRequired) Then
        ProgressBar1.Invoke(myDelegate)
    Else
        Count()
    End If
End Sub
```

Bien gérer les erreurs avec plusieurs processus

Nous allons maintenant aborder un point délicat (un de plus) dans la gestion multithread : la gestion des erreurs. Dans notre application, nous nous sommes arrangés pour qu'il n'y ait pas d'erreur, mais ce ne sera pas toujours le cas.

Réutilisons notre application dans le cas où elle crée une erreur *Cross Thread Exception*, par exemple.

```
    Public Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
Dim thread1 As New Threading.Thread(AddressOf Count)
thread1.Name = "ThreadCount"
thread1.Start()
    End Sub
```

Maintenant, tentons d'éviter l'erreur en utilisant un bloc Try Catch.



*Reportez-vous à ce sujet au chapitre **Rendre un programme robuste**.*

```
Try
  Dim thread1 As New Threading.Thread(AddressOf Count)
  thread1.Name = "ThreadCount"
  thread1.Start() 'demarre la 1ère opération dans un thread
  Catch ex As Exception
  Console.WriteLine("il y a eu une erreur dans le thread
                   secondaire")
End Try
```

L'utilisation d'un bloc `Try Catch` étant censée rattraper les erreurs, au pire, notre méthode `count` ne s'exécutera pas comme il faut, mais au moins elle ne plantera pas l'application.

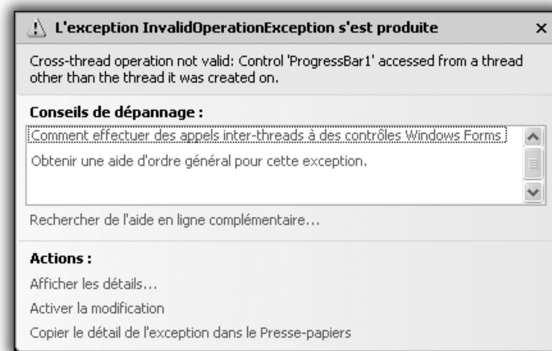


Figure 9.96 : *Exécution protégée par un Try Catch*

Perdu, l'application a quand même planté avec une erreur *Cross Thread Exception*. Pourquoi ?



Try et Catch

Un bloc `Try Catch` protège des exceptions survenant dans le thread qui a exécuté ce bloc `Try Catch`.

Dans notre cas, nous avons un bloc `Try Catch` qui fait appel à un autre thread. Il n'y a eu aucun problème dans notre thread principal. C'est notre thread secondaire, qui, en voulant modifier un élément graphique qu'il n'avait pas créé, a provoqué une erreur.

Très bien, protégeons donc notre second thread en mettant un bloc `Try Catch` dans notre méthode `count`, et exécutons de nouveau :

```
Public Sub Count()
    Try
        Dim count As Integer = Integer.Parse(TextBox1
      ✂ .Text)
        ProgressBar1.Maximum = count


        TextBox2.Text = "Launching count..."

        For i As Integer = 1 To count
            ProgressBar1.Value = i
            TextBox2.Text = "" & i.ToString()
        Next i

        TextBox2.Text = "Count finished."
    Catch ex As Exception
        Console.WriteLine("il y a eu une erreur
      dans le thread secondaire")

    End Try
End Sub
```

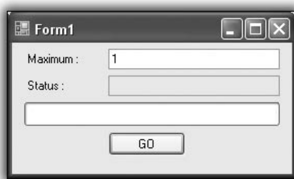


Figure 9.97 :
Thread secondaire protégé par un bloc Try Catch

Il ne s'est rien passé. Rien de visible, en tout cas. De plus, Visual Studio nous a bien fait état d'une erreur *Cross Thread Exception*. Cependant, l'application n'a pas planté, elle est toujours active, ce qui n'était pas le cas précédemment. Voyons la fenêtre de sortie.

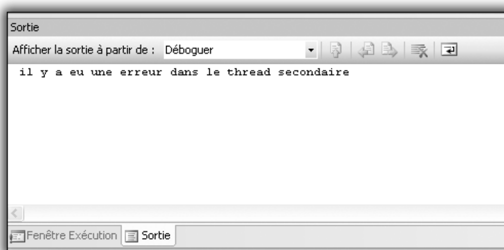


Figure 9.98 :
Sortie du thread secondaire protégé

Nous sommes passés dans la partie `Catch`. C'est une bonne nouvelle, car cela nous montre bien que le thread secondaire a été protégé. Ceci nous amène a une conclusion importante.



Protection sur les erreurs

Tous les threads de votre programme doivent gérer la protection de leurs erreurs.

Comme nous avons pu le constater tout au long de ce chapitre, la gestion d'un contexte multithread est une chose très subtile. Cependant, elle donne à vos applications une nouvelle dimension, car elle permet de bénéficier d'une activité accrue, ainsi que profiter du maximum de ressources présentes sur le système. Cela demande en contrepartie une rigueur et une attention particulières. Ne vous inquiétez pas si cela vous pose des soucis, car même des professionnels se trouvent régulièrement confrontés à des erreurs qu'ils ne comprennent pas ou qu'ils ont du mal à solutionner à cause des threads. Cela demande également de la pratique et de la curiosité. N'hésitez pas à jouer avec les threads, à comparer, à en ralentir exprès avec la commande `Sleep` pour voir le comportement, mais tentez aussi de diviser vos applications en sous-traitements exécutables par plusieurs threads. Elles prendront tout de suite une autre dimension, croyez-moi. Bon courage !

Chapitre 10

Valider les acquis

Vous avez à présent tous les outils en main pour devenir de bons programmeurs. Il ne vous manque plus que la pratique. Comme un musicien, il faut jouer le plus possible de votre instrument. Vous avez vu pour le moment des exemples simples à réaliser et conçus en moins de 15 minutes. À partir de maintenant, vous créerez des programmes plus complexes, plus longs à concevoir et qui mettent en jeu toutes les notions acquises. Il n'y a aucune limite aux applications que vous pouvez produire.

Vous allez dans ce chapitre réaliser deux applications dont vous pourrez vous servir tous les jours. La première est un album photo, qui va vous permettre d'afficher vos photographies en taille réelle, de les faire pivoter, etc. Ensuite, nous introduirons les notions de référence et d'importation, qui permettent d'utiliser du code déjà écrit et compilé.

Puis, nous parlerons de la bibliothèque DirectX, qui permet des manipulations multimédias. Ensuite, vous créerez un lecteur multimédia grâce auquel vous pourrez écouter de la musique et lire des vidéos.

10.1. Un album photo

Cette section intéressera les possesseurs d'appareils photos numériques et plus largement tous ceux dont les précieux moments de la vie ont été numérisés sous forme de fichiers image. Si vous avez déjà connecté votre appareil photo à votre ordinateur, vous avez pu remarquer que le format des photos était généralement JPG. Il produit des documents de taille réduite, ce qui facilite leur exploitation. Vous allez donc réaliser un album photo numérique qui proposera quelques fonctions simples de manipulation et de copie des images dans un dossier.

Le contrôle ListView

Lancez Visual Basic et créez un nouveau projet que vous appellerez `Album Photo` (voir Figure 10.1).

1 Dans le designer de formulaires, ouvrez la boîte à outils et ajoutez au formulaire principal un contrôle *ListView*.

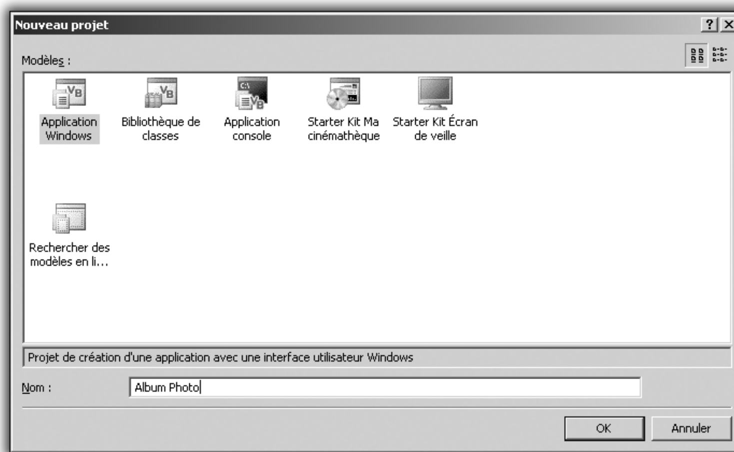


Figure 10.1 : *Projet d'album photo*



Listview

La *ListView* permet, un peu à la manière du contrôle *TreeView* vu précédemment, d'afficher une collection d'objets.

Pour le moment la *ListView* ressemble à un rectangle blanc. C'est normal. Ancrez-la à gauche du formulaire de manière qu'elle prenne toute la hauteur du formulaire.

2 Utilisez la propriété *Dock* en mettant la valeur à *Left*, de sorte que la *ListView* occupe la gauche du formulaire.

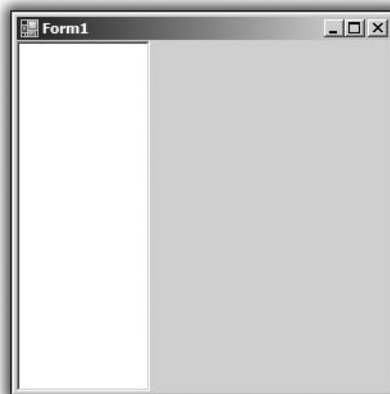


Figure 10.2 : *La ListView*

3. Ajoutez au formulaire un contrôle de type *FolderBrowser* en le faisant glisser depuis la boîte à outils. Vous allez utiliser ce contrôle pour qu'au chargement de l'application, il vous soit demandé où sont stockées les photos à afficher.
4. Ajoutez une *PictureBox* au formulaire. Elle servira à visualiser les images sélectionnées.
5. Une fois la *PictureBox* ajoutée, double-cliquez sur la barre de titre du formulaire, pour vous placer dans l'éditeur de code à l'endroit de l'événement correspondant au chargement du formulaire.

Avant d'entrer les instructions, prenez quelques minutes pour réfléchir à ce que vous faites. De quoi avez-vous besoin ? Une fois que l'utilisateur a choisi le dossier contenant ses images, que faut-il faire ? Il faut commencer par vérifier le type des fichiers. Pour cela, vous allez récupérer, pour chaque fichier présent dans le répertoire de l'utilisateur, son extension, et vous assurer qu'il s'agit d'un fichier JPG.

Cette réflexion conduit au code suivant :

```
If FolderBrowserDialog1.ShowDialog = _
 Windows.Forms.DialogResult.OK Then

For Each fichier As _
String In _
System.IO.Directory.GetFiles( _
FolderBrowserDialog1.SelectedPath)

If System.IO.Path.GetExtension(fichier) = ".jpg" Then
```

La première ligne correspond au test. Est-ce que l'utilisateur a bien cliqué sur le bouton OK pour valider le choix de son répertoire ? Si oui, vous utilisez une boucle de type `For Each` pour vérifier chaque fichier contenu dans le dossier accessible via le chemin donné par l'utilisateur. La vérification se fait à l'aide de la méthode `GetFiles` de l'espace de noms `System.IO.Directory`, qui est un ensemble de méthodes et de fonctions destinées à faciliter la manipulation des fichiers et dossiers.

La propriété `FolderBroserDialog.SelectedPath` renvoie le chemin complet du dossier sélectionné, sous forme de chaîne de caractères. En la passant en paramètre de la méthode `GetFiles`, vous recevez un tableau de chaînes de caractères dont chaque entrée correspond à un fichier du répertoire.

Une fois que vous avez ce tableau, il faut tester chaque entrée pour vérifier si oui ou non son extension est bien *.jpg*. Pour cela, vous utilisez une autre méthode de l'espace de noms `System.IO.Path.GetExtension`. Lorsque vous lui donnez en paramètre le chemin d'accès au fichier, elle compare l'extension de ce dernier avec *.jpg*. Il reste alors à écrire la suite du test conditionnel. Que se passe-t-il si le fichier possède bien une extension de type JPG ?

Vous allez tout simplement l'ajouter à la *ListView*. Il est temps de s'intéresser à ce nouveau contrôle.

6 Retournez dans le designer de formulaires. Sélectionnez la *ListView* et affichez la liste de ses propriétés. Cherchez la propriété nommée *View*. Elle représente la manière dont vont être vus les éléments de la *ListView*. Choisissez *LargeIcons*.

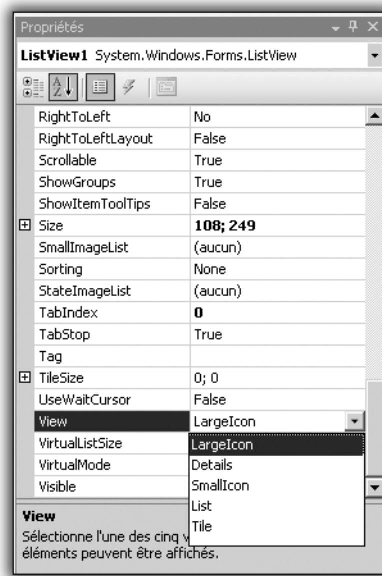


Figure 10.3 : *Modification de la propriété View*

Comme pour le contrôle *Listbox*, chaque élément de la *ListView* est un *Item* et possède des propriétés et des méthodes qui vont vous faciliter la tâche.

7 Revenez à l'éditeur de code. Vous avez la condition de test, il faut remplir les instructions si le test réussit.

Pour chacun des fichiers du répertoire de type JPG, il va falloir un objet de type *Item* correspondant dans la *ListView*.

8 Si le test réussit, vous créez un objet de type *ListViewItem* qui va contenir l'image.

```
Dim MonImage As ListViewItem = ListView1.Items
✂ .Add(fichier)
MonImage.Name = fichier
MonImage.Text = System.IO.Path
✂ .GetFileNameWithoutExtension(fichier)
End If
```

La première chose à faire est de déclarer une nouvelle variable de type *ListViewItem* (voir la première ligne). Dans la mesure où il est nécessaire d'initialiser cette variable, vous faites d'une pierre deux coups en l'initialisant avec la chaîne de caractères que représente la variable `Fichier` et vous l'ajoutez directement à la collection *Items* de la *ListView* avec la méthode `Add()`.

La deuxième ligne permet de faire un lien entre le fichier sur l'appareil photo ou le disque dur et le nom de l'objet *ListViewItem* dans le contrôle *ListView*. Ce nom permet également d'identifier l'image de manière unique. Une fois ajouté cet objet à la *ListView*, vous allez modifier sa propriété *Text*, pour afficher un nom de photo plus facile à retenir que le chemin complet. La propriété *GetFileNameWithoutExtension* permet de récupérer uniquement le nom du fichier dont le chemin est passé en paramètre.

N'oubliez pas de fermer la boucle conditionnelle, ce qui est fait en dernière ligne.

9 Lancez l'application avec la touche F5.

10 Au lancement, la sélection du répertoire contenant les photos est proposée. Sélectionnez un répertoire contenant des images au format JPG, et cliquez sur le bouton OK (voir Figure 10.4).

Si vous n'avez pas d'image à votre disposition, vous pouvez en télécharger à l'adresse **www.flickr.com/photos/splashworld/**.

La *ListView* devrait être remplie par les noms de tous les fichiers JPG. Cela dit, afficher le nom des images, c'est bien, mais afficher les images, ce serait mieux !

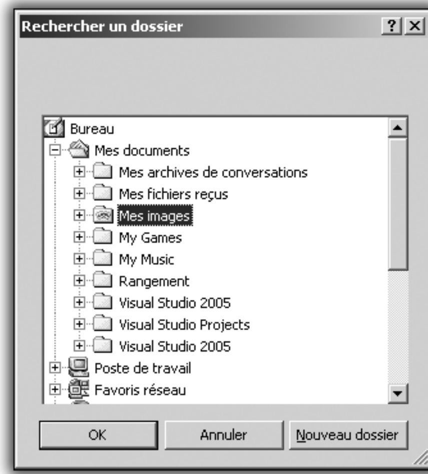


Figure 10.4 : *Répertoire d'images*

11 Quittez l'application et revenez au designer de formulaires.

12 Sélectionnez le contrôle *ListView* et affichez la fenêtre des propriétés, pour modifier les événements. Double-cliquez sur la case vide à droite de l'événement "clic" de la *ListView*. Une fois dans l'éditeur de code, entrez l'instruction suivante :

```
PictureBox1.Image = Image.FromFile(ListView1
✂ .SelectedItems.Item(0).Name)
```

Cette instruction va générer en mémoire une image à partir du chemin du fichier associé à l'élément de la *ListView* actuellement sélectionné. Dans la mesure où plusieurs objets de la *ListView* peuvent être sélectionnés, il faut afficher dans la *PictureBox* uniquement le premier d'entre eux. Cela explique l'utilisation de l'index 0 de la collection *SelectedItems*.

13 Relancez l'application avec F5.

14 Lorsque vous cliquez sur un élément de la *ListView*, il s'affiche dans la *PictureBox* (voir Figure 10.5).

15 Pour le moment tout se passe bien. Mais si au cours de l'exécution, il vous prend l'envie de changer de répertoire pour visualiser d'autres photos ? Dans la situation actuelle, il faudrait redémarrer l'application pour choisir un nouveau dossier, ce qui est fort peu pratique. Ajoutez un contrôle *Button* au formulaire dans le designer, puis double-cliquez dessus pour vous placer dans le code de l'événement "clic" du bouton.

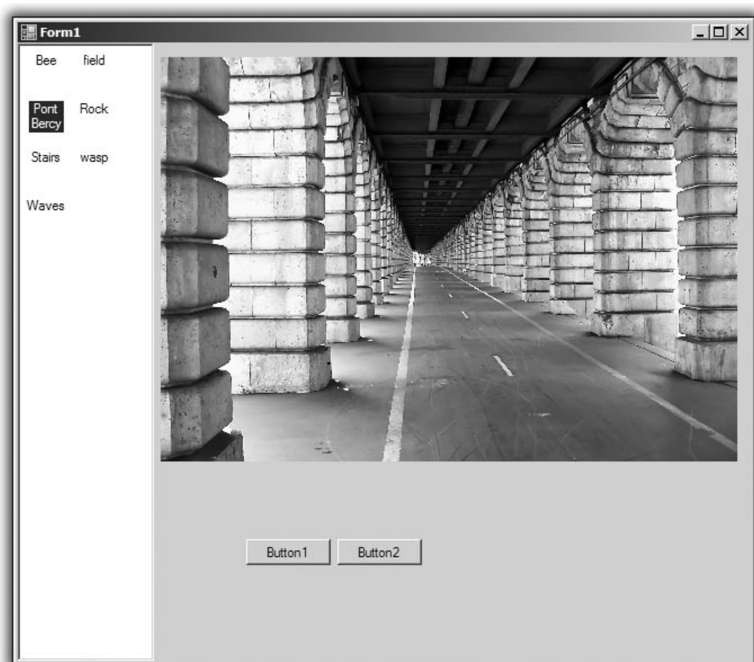


Figure 10.5 : Application et image chargée

Vous avez déjà un contrôle *FolderBrowserDialog* dans le projet, il est donc inutile d'en ajouter un autre.

16 Recopiez le code suivant, qui a pour but de réutiliser tous les contrôles existants.

```
If FolderBrowserDialog1.ShowDialog = _
Windows.Forms.DialogResult.OK Then
For Each fichier As String In _
System.IO.Directory.GetFiles( _
FolderBrowserDialog1.SelectedPath)
  If System.IO.Path.GetExtension(fichier) = _
".jpg" Then
  Dim MonImage As ListViewItem = _
ListView1.Items.Add(fichier)
                    MonImage.Name = _
fichier
                    MonImage.Text = _
System.IO.Path.GetFileNameWithoutExtension(fichier)
   End If
  Next
End If
```

Vous réalisez ici les mêmes vérifications et les mêmes allocations que lors du lancement du programme. Chaque image du nouveau répertoire est ajoutée à la *ListView*.

Maintenant que vous avez un programme simple et complet, compliquez-le un peu en lui ajoutant des fonctionnalités. Par exemple, dans la colonne des noms des photos, insérez un aperçu des documents.

En l'état actuel des choses, il paraît difficile de remplacer le nom de l'image par une version miniature de l'image elle-même. La première idée qui viendrait à l'esprit serait d'ajouter une *PictureBox* dans le contrôle *ListView*, et ce pour chaque image qu'elle contient. D'une part, cela ne fonctionne pas, et d'autre part, il est impossible de connaître à l'avance le nombre de photos qu'un utilisateur aura dans un dossier. Ce n'est donc pas une bonne idée…

Le contrôle ImageList

La bonne méthode consiste à utiliser le contrôle *ImageList*.



ImageList

Une *ImageList* est un contrôle qui permet de constituer une liste d'images. Vous pouvez utiliser cette liste avec d'autres contrôles contenant des collections d'éléments pour mettre en relation un objet et une image de la liste. Cette mise en relation est faite par un index qui caractérise la position de l'image dans la liste. Les images de la liste constituant une collection, l'index du premier élément est 0.

Comme les éléments que vous ajoutez dans la *ListView* possèdent également un index, il suffit de synchroniser les ajouts dans la *ListView* et dans l'*ImageList* pour qu'un élément ait le même index dans les deux collections. Passons à la pratique.

1 Commencez par ajouter au formulaire un contrôle *ImageList*.

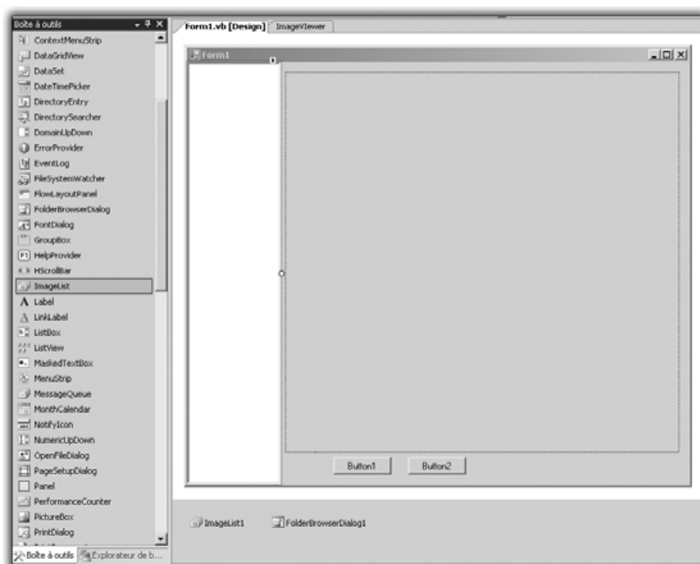


Figure 10.6 : *Ajout d'un contrôle ImageList*

Retournez au code de chargement de l'application en double-cliquant sur la barre de titre dans le designer de formulaires et remplacez le code par le suivant :

```
If FolderBrowserDialog1.ShowDialog = Windows.Forms
✂ .DialogResult.OK Then
For Each fichier As String In System.IO.Directory
✂ .GetFiles(FolderBrowserDialog1.SelectedPath)
If System.IO.Path.GetExtension(fichier) = ".jpg" Then
Dim MonImage As ListViewItem = ListView1.Items
✂ .Add(fichier)
MonImage.Name = fichier
MonImage.Text = System.IO.Path
✂ .GetFileNameWithoutExtension(fichier)
ImageList1.Images.Add(Image.FromFile(fichier))
MonImage.ImageIndex = MonImage.Index
End If
Next

End If
```

Les deux nouvelles instructions nécessaires à la synchronisation de l'image et de son aperçu sont les deux dernières lignes de la boucle `For Each`.

```
ImageList1.Images.Add(Image.FromFile(fichier))
MonImage.ImageIndex = MonImage.Index
```

La première ajoute une image à l'*ImageList* en utilisant la méthode `FromFile` de l'objet `Image`. Ce dernier est fourni par le Framework .NET 2.0, qui permet la manipulation d'images, ou encore comme ici, le chargement d'une image à partir du chemin d'un fichier.

La deuxième donne à l'image dans la *ListView* un `ImageIndex`, qui est un nombre entier représentant l'image à utiliser dans l'*ImageList*.

Dans ce cas, l'image est ajoutée simultanément dans l'*ImageList* et dans la *ListView*, l'index est donc le même. C'est pourquoi vous associez le numéro d'index `MonImage.Index` à la propriété `ImageIndex`.

Une question se pose alors. Si vous pouvez ajouter une image dans la *ListView*, pourquoi garder la *PictureBox* sur le formulaire ? Simplement parce que l'*ImageList* n'est faite que pour afficher des miniatures. La *PictureBox* dispose de contrôles plus appropriés à l'affichage d'images.

2 Revenez au designer de formulaires et sélectionnez la *ListView*.

3 Affichez la fenêtre des propriétés et cherchez la propriété nommée *LargeImageList*.

4 Cliquez sur le menu déroulant à droite et choisissez *ImageList1*, qui représente la liste.

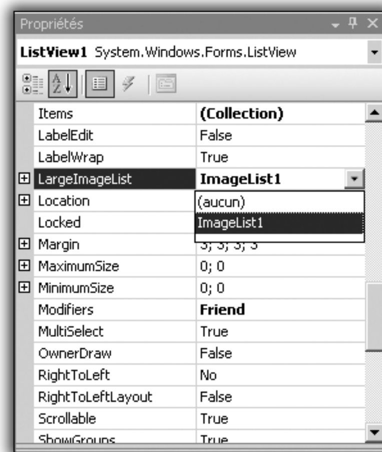


Figure 10.7 : *Modification de la propriété LargeImageList*

5 Lancez l'application. Sélectionnez le répertoire d'images, puis cliquez sur OK. Normalement le programme met légèrement plus longtemps pour se charger. Cela est dû au fait qu'il faut charger les miniatures de chaque image dans la mémoire de votre

ordinateur une bonne fois pour toutes. Ensuite, le programme affiche les miniatures des images à gauche, et à droite la *PictureBox* contenant l'image sélectionnée.

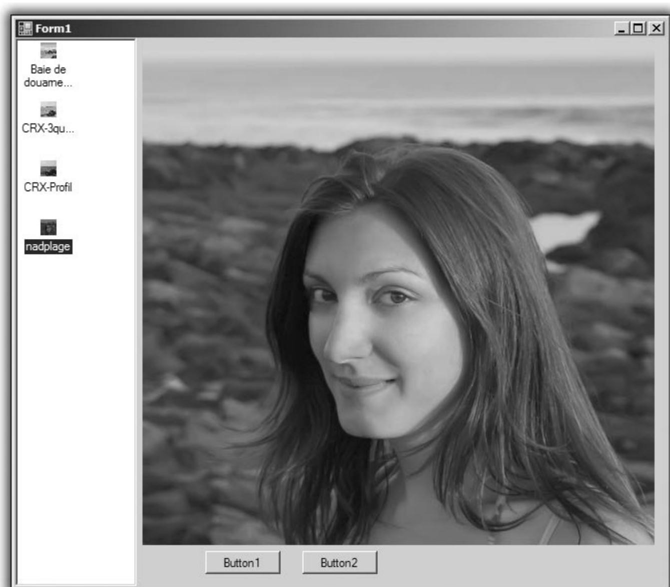


Figure 10.8 : *Les miniatures*

Si vous désirez changer la taille des miniatures, retournez dans le designer de formulaires, puis cliquez sur *ImageList1* en bas du designer. Affichez la fenêtre des propriétés puis cherchez la propriété nommée *ImageSize*. La taille par défaut est initialisée à *16;16*.

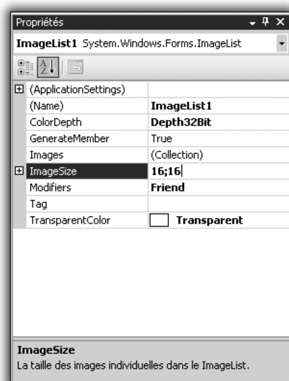


Figure 10.9 : *Taille des miniatures*

Changez ces deux valeurs, qui représentent respectivement la hauteur et la largeur des miniatures. Par exemple, inscrivez `128;128`, en insérant le caractère de séparation qui est un point-virgule.



Les couleurs

Pour que les miniatures soient plus belles, vous pouvez changer la profondeur de l'espace couleur utilisé pour les représenter en modifiant la propriété *ColorDepth* du contrôle *ImageList*. Attention, les performances de l'application sont directement liées à la taille des ressources qui lui sont allouées. En clair, plus les miniatures seront grandes, plus l'application mettra de temps à se charger. Si vous êtes photographe professionnel et que les répertoires contiennent plusieurs centaines d'images, le chargement des miniatures à chaque changement de répertoire risque de devenir handicapant.

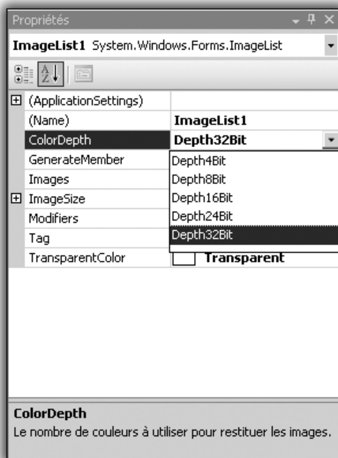


Figure 10.10 : *Changement de la profondeur des couleurs*

Lancez l'application avec F5 pour constater les changements (voir Figure 10.11).

Il est temps à présent de tester toutes les possibilités du programme. Cliquez sur le **bouton 1** et sélectionnez un nouveau répertoire pour en afficher les images. Remarquez que les images ne viennent pas remplacer celles déjà présentes, mais s'ajoutent à la liste d'images. Cette fonctionnalité peut se révéler intéressante, mais ce n'est pas exactement ce que vous voulez. Transformer un mauvais comportement d'un logiciel en une fonctionnalité supplémentaire est une mauvaise habitude ! Par conséquent, corrigez ce comportement.



Figure 10.11 : *Des miniatures grand format*

Le problème vient du fait que lorsque vous modifiez le code exécuté lors du clic sur le bouton est le même que lors du chargement du programme. Allez dans l'éditeur de code en double-cliquant à partir du designer. Remplacez le code par celui-ci :

```
If FolderBrowserDialog1.ShowDialog = _
Windows.Forms.DialogResult.OK Then
            ListView1.Items.Clear()
            ImageList1.Images.Clear()
            For Each fichier As String In _
System.IO.Directory.GetFiles( _
FolderBrowserDialog1.SelectedPath)
   If System.IO.Path.GetExtension(fichier) _
= ".jpg" Then
  Dim MonImage As ListViewItem = _
ListView1.Items.Add(fichier)
                    MonImage.Name = fichier
                    MonImage.Text = System.IO.Path
                    ✂ .GetFileNameWithoutExtension(fichier)
   ImageList1.Images.Add(Image.FromFile(fichier))
   MonImage.ImageIndex = MonImage.Index
                End If
            Next
        End If
```

La différence se situe juste après le clic sur le bouton OK. Deux instructions sont ajoutées :

```
ListView1.Items.Clear()
ImageList1.Images.Clear()
```

La première vide tous les éléments de la *ListView*. La deuxième répète l'opération, mais avec la liste d'images comme cible. Une fois cela fait, les deux contrôles récupèrent la liste des images qu'ils doivent enregistrer.

L'ajout marche parfaitement. Si vous le souhaitez, reprenez le code sans les modifications et placez-le dans un autre bouton. Vous aurez ainsi un bouton d'ajout d'image disponible lors de la séance de visualisation en cours et un bouton de vidage des listes.

Vous avez maintenant une application de visualisation d'images presque complète. Vous allez ajouter une fonctionnalité appréciée : la rotation d'image.

Faire pivoter une image

Vous avez une *PictureBox*, qui contient une image. Malheureusement, ce contrôle ne dispose pas de méthode pour faire pivoter une image. Il va falloir ruser.

1 Commencez par ajouter au projet un deuxième formulaire. Pour cela, choisissez **Ajouter un nouvel élément** dans le menu **Projet**.

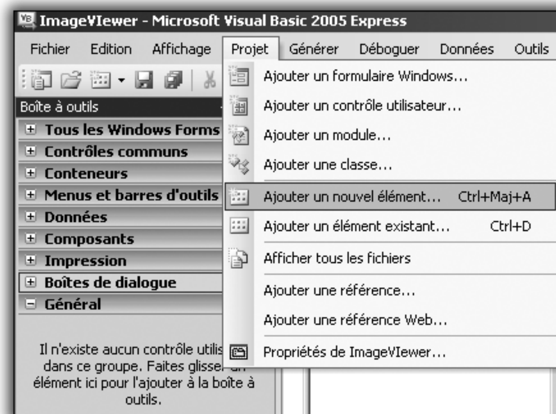


Figure 10.12 : *Ajout d'un nouvel élément*

Puis choisissez un nouveau formulaire (*Windows Form*).

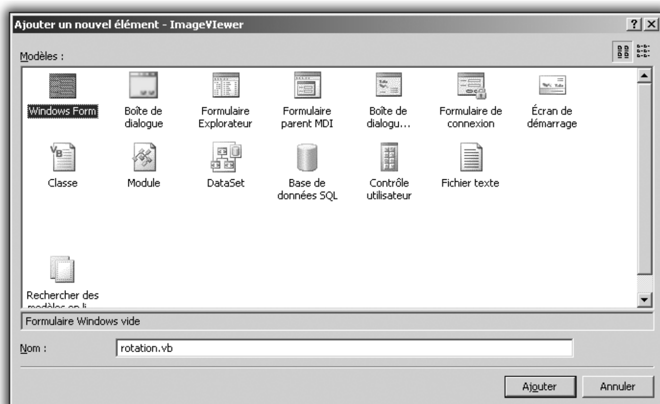


Figure 10.13 : *Choix du formulaire*

Dans le designer de formulaires, ajoutez un contrôle *PictureBox*, destiné à accueillir l'image une fois la rotation effectuée.

2 Une fois la *PictureBox* ajoutée, sélectionnez-la et affichez ses propriétés.

3 Dans les événements, repérez celui qui est nommé *Paint*.

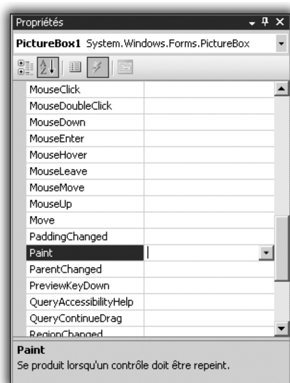


Figure 10.14 : *Modifier l'événement Paint*



Événement Paint

Il correspond au moment où la *PictureBox* est dessinée sur le formulaire.

Vous allez donner les instructions pour que l'image à afficher subisse une rotation de 90 degrés dans le sens des aiguilles d'une montre.

4 Dans l'éditeur de code de l'événement *Paint*, entrez les instructions suivantes :

```
Dim img As Image = Form1.PictureBox1.Image
Dim g As Graphics = e.Graphics

Dim M As New Matrix
M.RotateAt(90, New PointF(Width / 2, Height / 2))
g.Transform = M
g.DrawImage(img, 0, 0)
```

La première ligne sert à récupérer l'image qui est affichée dans le premier formulaire, donc l'image courante. Ensuite, il faut initialiser une variable de type `Graphics`. Elle va servir à dessiner sur l'espace de la *PictureBox*. C'est la ruse dont il était question plus haut. La *PictureBox* ne permet pas directement la modification des images, il faut donc passer l'image courante en mémoire, la modifier, et faire appel au moteur de dessin des formulaires pour afficher la version modifiée.

L'image est à présent en mémoire et vous pouvez dessiner sur sa surface. Il faut faire pivoter le document. Pour cela, il faut passer par une matrice de transformation. Rassurez-vous, il est inutile de ressortir vos cours de mathématiques du lycée, tout est automatisé. Il faut déclarer une variable de type `Matrix` qui va servir de support à la transformation. Pour déclarer un objet de type `Matrix`, vous devez effectuer une importation de l'espace de noms `System.Drawing.Drawing2D`.

Pour cela, ajoutez la ligne suivante tout en haut du fichier du formulaire dans l'éditeur de code (elle doit devenir la première ligne du fichier) :

```
Imports System.Drawing.Drawing2D
```

Cela fait, vous déclarez une rotation de la matrice `M` à l'aide de la méthode `Rotate`.

```
M.RotateAt(90, New PointF(Width / 2, Height / 2))
```

Cette instruction précise qu'une rotation est effectuée. Les paramètres de la méthode définissent l'angle (ici 90 degrés) et l'origine de la rotation (ici le centre de la *PictureBox*).

Une fois la matrice définie, il faut préciser au programme qu'il s'agit de la matrice de transformation de l'espace graphique `G`, ce qui se fait par la ligne suivante :

```
g.Transform = M
```

La dernière ligne fait appel à la méthode `Draw` de l'élément `Graphics`, qui permet d'effectuer réellement le dessin.

5 Il reste à appeler le deuxième formulaire à partir du premier. Revenez au designer de votre premier formulaire en cliquant sur celui-ci dans l'Explorateur de solutions. Ajoutez un contrôle *Button* et double-cliquez sur ce dernier pour vous placer dans l'éditeur.

6 Ajoutez le code suivant :

```
Dim form2 As Rotation90
form2 = New Rotation90
form2.Show()
```

Dans cet exemple, le deuxième formulaire du projet est renommé en *Rotation90*, ces deux lignes ayant pour but de générer une fenêtre de ce type suite à un clic du **bouton 2**.

Lancez l'application en appuyant sur F5. Choisissez une image dans la *ListView* de gauche, puis cliquez sur le **bouton 2**. L'image ayant subi une rotation de 90 degrés vers la droite s'affiche dans un nouveau formulaire.

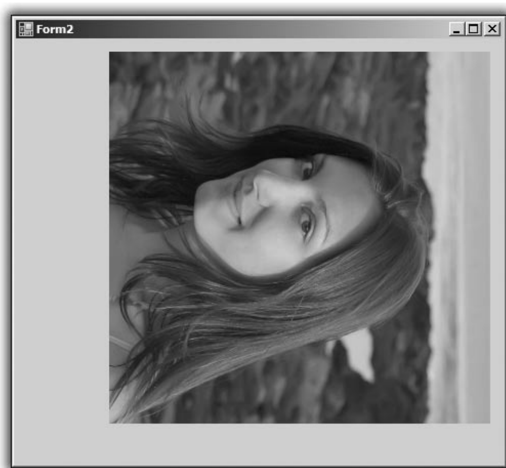


Figure 10.15 : *Image avec rotation*

Vous pouvez reprendre ce principe et ajoutez d'autres boutons de rotation qui correspondrait aux angles de –90 et 180 degrés. Vous pouvez même imaginer une *TrackBar* qui permette de définir l'angle de la rotation.

7 La dernière fonction à ajouter est la fermeture simplifiée du deuxième formulaire. Dans le designer du deuxième formulaire, double-cliquez sur la *PictureBox*. Une fois dans l'éditeur, ajoutez la ligne suivante :

```
Me.Close()
```

Cela aura pour effet de fermer le formulaire quand vous cliquerez sur l'image. Vous verrez qu'à la longue, c'est pratique !

10.2. Découvrir DirectX

Vous allez passer à un type de programmation un peu plus ludique, puisque relative au multimédia, c'est-à-dire à la vidéo et à la musique.

DirectX est un ensemble d'outils qui était, à ses débuts, destiné aux développeurs de jeux vidéo. Il s'agit de fonctions et de bibliothèques de fonctions qui permettent de gérer l'affichage de données 3D, 2D, de jouer des sons, des vidéos, de capturer les événements du clavier, etc.

Dans cet exemple, vous allez développer un lecteur multimédia. La première chose à faire est de télécharger le kit de développement DirectX à l'adresse http://msdn.microsoft.com/directx/sdk/. Téléchargez la version la plus récente, puis lancez le programme d'installation. Ensuite, vous pourrez utiliser les fonctions de DirectX à travers Visual Basic Express.

10.3. Un lecteur multimédia

Une fois le kit de développement installé, créez un nouveau projet Visual Basic que vous intitulerez `MyPlayer`. Ce programme vous servira à lire de la musique et des vidéos. Vous allez ajouter une référence permettant d'utiliser les fonctions audio et vidéo de DirectX.

1 Dans l'Explorateur de solutions, cliquez du bouton droit sur le nom du programme et sélectionnez **Ajouter une référence**.

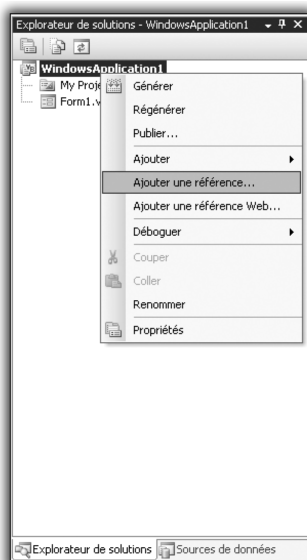


Figure 10.16 :
Ajout d'une référence

La boîte de dialogue suivante s'affiche :

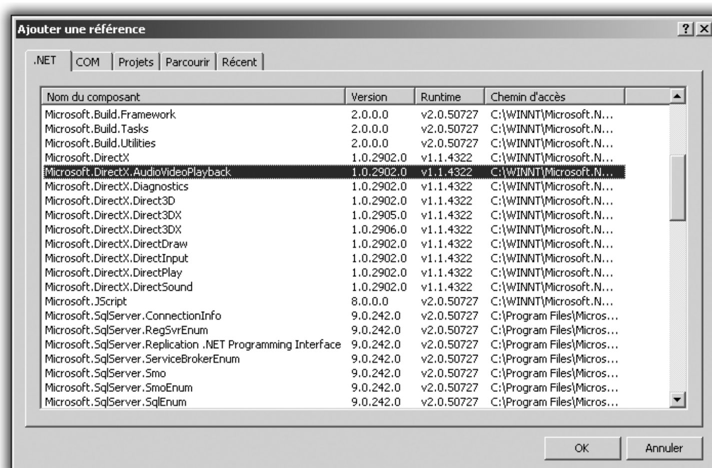


Figure 10.17 : *Fenêtre d'ajout de référence*

2 Dans la longue liste des références disponibles, sélectionnez *Microsoft.DirectX.AudioVideoPlayback*. Vous pourrez ainsi utiliser les fonctions permettant de diffuser des fichiers audio et vidéo. Ajoutez un menu à l'application.

3 Ajoutez un bouton et appelez-le `Jouer un fichier audio`.

4 Double-cliquez sur le bouton et une fois dans l'éditeur, ajoutez le code suivant :

```
If OpenFileDialog1.ShowDialog = Windows.Forms
✂ .DialogResult.OK Then
       Dim PlayAudio As New Microsoft.DirectX
       .AudioVideoPlayback.Audio(OpenFileDialog1
       ✂ .FileName)
       PlayAudio.Play()
End If
```

Ce code déclare un nouvel objet de type audio, puis le joue. Pour le moment, vous ne pouvez pas stopper la lecture du morceau. Pour ce faire, créez votre propre classe. Implémentez ensuite les méthodes appropriées.



*Lisez à ce sujet le chapitre **Passer au niveau supérieur**.*

Le fichier se joue via un appel à la méthode `Play()` de l'objet `audio`. Une méthode `Stop()` permet d'arrêter la lecture en cours.

Passons à la vidéo. La lecture de la vidéo se faire de la même manière.

5 Ajoutez un bouton au formulaire et éditez son code. Insérez les lignes suivantes :

```
If OpenFileDialog1.ShowDialog = Windows.Forms
✂ .DialogResult.OK Then
    Dim PlayVideo As New Microsoft.DirectX
    .AudioVideoPlayback.Video(OpenFileDialog1
    ✂ .FileName)
PlayVideo.Play()
End If
```

La vidéo se lance dans une fenêtre à part. Comme pour l'audio, il n'est pas possible de la stopper, à moins de réaliser une classe à part dans laquelle vous gérerez toutes les propriétés d'une vidéo.

Vous pouvez ajouter des fonctionnalités au lecteur multimédia. Par exemple, l'ajout d'une *ListBox* permettrait l'utilisation d'une liste de lecture. Si vous reprenez la méthode de lecture des fichiers d'un répertoire de l'exemple précédent, vous pouvez jouer tout le contenu multimédia d'un répertoire ! Rappelez-vous que les seules limites du programme sont celles de votre imagination.

Chapitre 11

Programmer pour le Web

Vous avez appris à réaliser des applications Windows. Vous allez voir comment tous les concepts peuvent s'appliquer à la réalisation de sites Internet. En lisant ce chapitre, vous serez à même de mettre en ligne votre propre blog.

Vous apprendrez à réaliser une page web de manière classique, puis à lui ajouter du contenu dynamique pour finir par mettre en pratique tout ce que vous avez déjà vu sur les applications Windows pour créer une interface riche sur votre site.

11.1. Le langage HTML

À la base des sites Internet, il y a le langage HTML (HyperText Markup Language ou "langage de balises hypertextes" en français). La première particularité du langage HTML est de ne pas avoir besoin d'être compilé. En effet, il est interprété par une application tiers, un navigateur web par exemple. Commençons par une page web simple.

```
<HTML>
<HEAD>
<TITLE>Ma première page</TITLE>
</HEAD>
<BODY>
Bonjour tout le monde
</BODY>
</HTML>
```

Copiez ces quelques lignes dans votre éditeur de texte favori et enregistrez le fichier en lui donnant l'extension *.html*. Vous précisez ainsi à votre système d'exploitation que le fichier que vous venez d'enregistrer est une page web.

Si vous double-cliquez sur ce fichier, il s'ouvrira dans votre navigateur web en affichant "Bonjour tout le monde" (voir Figure 11.1).

Quand vous ouvrez le fichier, votre navigateur fait une lecture ligne par ligne en interprétant les couples de balises placées dans le fichier.

Remarquez que chaque couple de balises est composé d'une balise ouvrante `<NomDeLaBalise>` et d'une balise fermante `</NomDeLaBalise>`. De cette manière, votre navigateur sait comment afficher ce qui se trouve entre les balises.

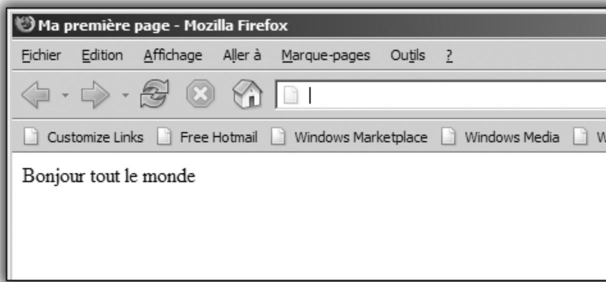


Figure 11.1 : *Votre première page*

Analysons maintenant le fichier dans le détail. Bien que la page affichée soit des plus élémentaires, elle respecte un format qui est celui de toutes les pages web du monde, et ce depuis plus de 30 ans.

La première ligne contient une balise `HTML`, qui spécifie au navigateur qu'il s'agit d'une page formatée dans ce langage. La particularité de cette balise est de délimiter la zone de la page qui sera affichée dans le navigateur. Normalement, tout ce qui est placé en dehors du couple de balise `HTML` n'est pas interprété par le navigateur.



Les navigateurs

Les navigateurs récents autorisent une certaine largesse vis-à-vis des conventions et affichent parfois le texte en dehors des balises. Ces tolérances ne sont pas forcément propres à tous les navigateurs. Attention donc à respecter les normes.

La deuxième balise, `HEAD`, sert à ouvrir une zone qui permet de définir un lot d'informations concernant la page. Ces informations ne sont pas nécessairement affichées. Dans cet exemple, vous ajoutez à l'intérieur des deux balises `HEAD`, deux balises `TITLE`. Ces deux balises permettent de définir un titre pour la page, qui sera affiché dans la barre de titre de la fenêtre. Ici c'est la chaîne de caractère "Ma première page". Une fois la balise `TITLE` fermée, il ne faut pas oublier de refermer la balise `HEAD` et ainsi de délimiter la fin de la zone d'en-tête. Une fois cela fait, vous avez spécifié toutes les informations nécessaires pour que le navigateur sache ce qu'il faut afficher.

Après l'en-tête de la page, il faut passer au corps. Pour préciser la zone de corps de page, il convient d'utiliser le couple de balises BODY. Entre ces balises, vous écrivez tout ce qui doit être affiché dans le navigateur. Il s'agit ici de la simple phrase "Bonjour tout le monde". Il convient ensuite de fermer la balise BODY dans la mesure où le corps de la page a fini d'être décrit, vu que vous n'avez plus rien à ajouter. Vous déclarez la fin du document HTML avec une balise fermante </HTML>.

11.2. Les liens HTML

Pour aller de page en page, vous pouvez ajouter des liens qui pointent vers une autre page à l'aide de la balise <A>. Elle produit un lien sur lequel l'utilisateur peut cliquer pour afficher dans le navigateur une autre page. Elle accepte un attribut HREF qui définit la page sur laquelle pointer. Voici une illustration de ce principe.

1 Créez une page web à l'aide du code suivant :

```
<HTML>
<HEAD>
    <TITLE>Ma page de lien</TITLE>
</HEAD>
<BODY>
<A HREF="./page2.html">Ma deuxième page web</A>
</BODY>
</HTML>
```

2 Créez ensuite une deuxième page, dans le même dossier que la précédente, avec le code suivant :

```
<HTML>
<HEAD>
    <TITLE>Ma page liée/TITLE>

</HEAD>
<BODY>
Bienvenue sur la deuxième page.
</BODY>
</HTML>
```

Lorsque vous afficherez la première page dans un navigateur, elle sera composé d'un lien qui vous mènera à la seconde.

Vous pouvez observer que le texte qui compose le lien est celui que vous entrez entre les balises <a> et .

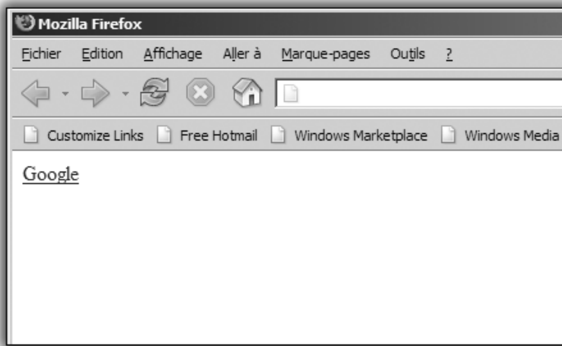


Figure 11.2 : *Une page avec un lien vers Google*

11.3. Ajouter du style

Vous avez vu comment ajouter du contenu à une page HTML. Mais votre page noire et blanche n'est pas attractive et vous allez l'agrémenter.

Le langage HTML a été inventé par des scientifiques pour la présentation de leurs rapports. Dans ce but, plusieurs balises ont été ajoutées au langage de manière à améliorer la mise en page. Vous pouvez ainsi mettre du texte en gras à l'aide des balises `<b>` et `</b>`, du texte souligné avec la balise `<u>` et `</u>`, ou encore du texte en italique avec la balise `<i>` et `</i>`. Si vous appliquez cela à la première page, vous obtenez :

```
<HTML>
<HEAD>
<TITLE>Ma première page</TITLE>
</HEAD>
<BODY>
<b>Bonjour</b> <u>tout</u> <i>le monde</i>
</BODY>
</HTML>
```

Il existe une longue liste de balises HTML dédiées aux styles, mais la fournir n'apporterait rien ici. Elles figurent sur le site web du W3C, **www.w3c.org**. Il s'agit de l'organisme qui gère les conventions et les normes pour toute la programmation web.

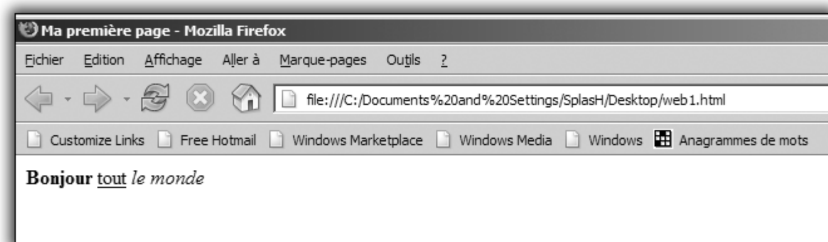


Figure 11.3 : *Une page qui a du style*

Notez qu'il est possible de combiner différents codes de mise en page. Vous pouvez ainsi écrire du texte gras souligné en faisant se chevaucher les balises. Mais attention à bien respecter leur ordre. Il faut toujours fermer en premier la balise que vous avez ouverte en dernier.

Code correct :

```
<b><i>Mon texte gras italique</i></b>
```

Code incorrect :

```
<b><i>Mon texte gras italique</b></i>
```

Cette méthode marche parfaitement mais est un peu lourde dès lors que l'on veut cumuler les attributs. C'est pourquoi, pour tout ce qui relève de la mise en page, il est maintenant courant d'avoir recours aux feuilles de style.

11.4. Garder le style

HTML dispose de balises destinées à la mise en page de contenus. Cependant, et pour plus de souplesse, sont apparues vers la fin des années 90 les feuilles de style (ou CSS). Elles permettent de séparer le contenu de sa présentation. En utilisant les CSS, vous aurez les instructions de mise en page réparties dans un fichier et le contenu du site dans un autre. De cette manière, vous pourrez réutiliser votre style sur autant de pages web que vous le souhaitez, sans avoir à récrire les instructions de mise en page. Autre avantage : si vous avez un piètre sens artistique, des centaines de feuilles de style sont librement téléchargeables sur Internet.

Avoir la classe

Voyons maintenant comment définir des attributs de style dans une feuille CSS. Une fois de plus, lancez votre éditeur de texte préféré et copiez les lignes suivantes :

```
.rouge
{
color:Red;
}
```

Remarquez bien le point (`.`) devant le mot `rouge`. Cela signifie que vous déclarez une classe. Cette classe, tout comme en programmation orientée objet, va permettre de définir un ensemble de propriétés. Une fois cela fait, vous pourrez appliquer toutes ces propriétés à une balise ou à un élément d'une page HTML en lui indiquant à quelle classe il doit se conformer.

Ici vous avez défini une classe `rouge` qui va contenir un seul attribut, celui de la couleur du texte. Tous les paramètres CSS sont déjà définis, il vous suffit juste de leur donner une valeur pour les utiliser. Ici vous utilisez l'attribut `Color`, qui permet de définir la couleur du texte dans un conteneur donné. Pour lui donner la valeur `rouge`, il suffit de noter :

```
color:Red
```

Ici, le caractère deux-points (`:`) permet de réaliser l'affectation.



Les attributs CSS

Les attributs CSS sont à peu près aussi nombreux que les balises HTML. C'est pourquoi nous n'en fournirons pas une liste détaillée ici. Vous pourrez les trouver sur Internet, en saisissant `Attributs CSS` dans votre moteur de recherche favori.

Enregistrez le fichier CSS en le nommant *fichierstyle.css* (notez l'extension *.css*). Sauvegardez-le si possible dans le même répertoire que le fichier HTML.

Vous allez maintenant indiquer à la page web qu'elle doit utiliser le style du fichier *.css*.

1 Ouvrez le fichier contenant le code HTML et ajoutez la ligne suivante entre les deux balises HEAD :

```
<link rel=stylesheet href=fichierstyle.css />
```

Cette balise définit un lien entre la page HTML et le fichier *.css*. Cela va permettre d'utiliser tous les styles définis dans le fichier.

2 Modifiez la balise BODY en ajoutant l'attribut class de la manière suivante :

```
<HTML>
<HEAD>
<TITLE>Ma première page</TITLE>
</HEAD>
<BODY class="rouge">
Bonjour tout le monde
</BODY>
</HTML>
```

À présent, tout le texte qui sera saisi entre les deux balises BODY sera de couleur rouge. Chargez votre page dans votre navigateur Internet. Le texte est maintenant devenu rouge.

Organiser la page

Voici le point méthodologique du chapitre. Comme vous l'avez vu dans la section précédente, vous pouvez facilement ajouter des attributs de style à une balise HTML. Cependant, comment avoir une partie du texte en rouge et une partie du texte en bleu ? Vous ne pouvez pas avoir deux ensembles BODY dans une page web. Il faut donc ruser. Il existe pour cela une balise qui permet de délimiter une zone à laquelle s'appliquera une série de paramètres. Il s'agit de la balise DIV. Elle vous permettra d'appliquer un style à une partie seulement de la page. Voici un exemple.

1 Ouvrez à nouveau le fichier *.css* et ajoutez la classe suivante :

```
.vert{
color:Green;
}
```

2 Ajoutez les balises suivantes au fichier HTML :

```
<HTML>
<HEAD>
<TITLE>Ma première page</TITLE>
```

```
</HEAD>
<BODY>
<div class="rouge">Bonjour tout</div>
<div class="vert">le monde</div>
</BODY>
</HTML>
```

3 Chargez la page dans un navigateur Internet.

Vous pouvez constater qu'à la sortie de chaque balise `DIV` vient s'ajouter un retour à la ligne. Pensez-y lors de la conception de vos pages web.

Prenez maintenant un moment pour tester les CSS que vous pourrez trouver sur Internet. Un bon endroit pour commencer vos recherches est le site **www.csszengarden.com**.

11.5. Faire bouger les pages

HTML est un langage descriptif et permet peu d'interaction avec les utilisateurs. Vous pouvez seulement leur présenter des données, plus ou moins bien agencées.

Pour pallier cette insuffisance, il existe plusieurs solutions, dont l'utilisation d'un deuxième langage dans une page web : JavaScript.

Un script est, comme un programme, une suite d'instructions exécutées les unes à la suite des autres, de manière séquentielle.

Les instructions JavaScript sont placées entre balises, de la manière suivante :

```
<script language="javascript">
document.writeln("Bonjour tout le monde");
</script>
```

Notez que ces lignes peuvent être placées à n'importe quel endroit de la page. Dans la balise ouvrante `<script>`, il est coutume de préciser le type de langage de script utilisé, ici JavaScript.

Une seule instruction figure entre les balises, qui va ici permettre d'écrire une ligne dans la page web. Recopiez les instructions précédentes entre les balises `BODY` de la page et chargez-la dans un navigateur web.

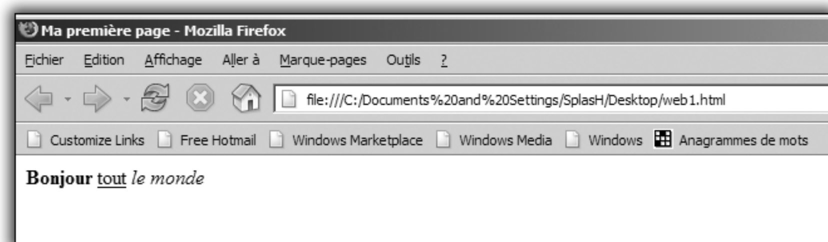


Figure 11.4 : La page résultat du script

Notez que vous recourez à une schématisation objet en utilisant la méthode `writeln` de l'objet `document`. En effet, en JavaScript, la page est schématisée sous forme d'objet, disposant de propriétés et de méthodes que vous pourrez modifier.

Voici par exemple une instruction qui permet de changer le titre de la page :

```
document.title="Ma page de script"
```

Placée entre les balises `script` de la page, celle-ci va modifier le titre par la chaîne de caractères placée après =.

JavaScript et les variables

Comme dans un programme, vous pouvez utiliser des variables pour améliorer vos pages. Il suffit de les déclarer en utilisant la directive `var` suivie d'un nom de variable.

Voici par exemple un script qui affichera le résultat de l'addition de deux variables :

```
<script language="javascript">
 var a = 3;
 var b = 4;
 document.writeln(a+b);
</script>
```

Une subtilité des variables JavaScript est qu'elles n'ont pas besoin d'être typées. En d'autres termes, vous n'avez en aucun cas à préciser que vous utilisez des entiers ou des chaînes de caractères par exemple.

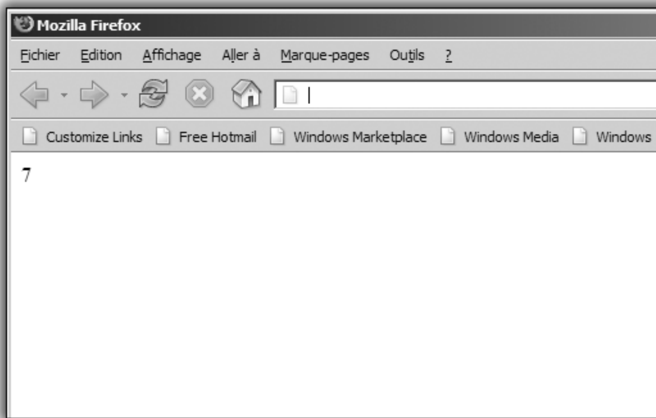


Figure 11.5 : *Résultat du script*

Remplacez ce dernier script par celui-ci :

```
<script language="javascript">
 var a = "3";
 var b = "4";
 document.writeln(a+b);
</script>
```

Au lieu d'afficher la somme de deux entiers, vous allez afficher la concaténation des deux chaînes de caractères, soit 34. On dit que JavaScript est faiblement typé. De la même manière, vous pouvez utiliser des nombres décimaux comme valeurs des variables :

```
<script language="javascript">
 var a = "3.5";
 var b = "4.5";
 document.writeln(a+b);
</script>
```



Les instructions

En JavaScript, chaque ligne d'instruction doit être terminée par un point-virgule. Si vous l'oubliez, le script ne sera pas interprété par le navigateur.

Rester fonctionnel

Vous pouvez également déclarer des fonctions en JavaScript, comme dans d'autres langages, de sorte à pouvoir réutiliser du code, clarifier le script et l'organiser de manière claire.

Pour déclarer une fonction en JavaScript, il convient d'utiliser le mot-clé `function` suivi du nom de la fonction. Voici un simple exemple qui affichera deux chaînes concaténées :

```
<script language="javascript">
 function add(){
 var a = "Bonjour";
 var b = "Tout le monde";
 return a+b;
 }
 document.writeln(add())

</script>
```

Pour qu'une fonction JavaScript retourne une valeur, il faut utiliser le mot-clé `return`.

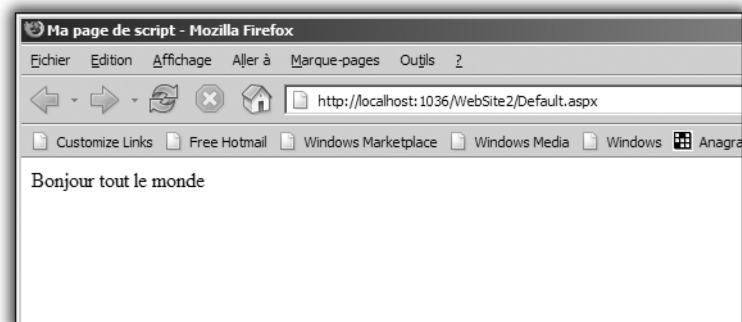


Figure 11.6 : *Les fonctions*

Les événements

Pour réellement rendre interactives vos pages, vous pouvez, tout comme avec Visual Basic, ajouter du code source lorsqu'un événement se produit sur la page. Vous avez vu pour le moment quelques balises HTML. Voyons maintenant comment ajouter un bouton sur la page HTML et exécuter du code lorsque l'utilisateur va cliquer dessus.

1 Ouvrez votre éditeur de texte et ajoutez la balise suivante :

```
<button onclick="coucou()">Bonjour</button>
```

2 Ajoutez le script suivant au-dessus de la balise qui définit le bouton :

```
<script language="javascript">
function coucou(){
window.alert("bonjour tout le monde");
}
</script>
```

Lorsque vous cliquez sur le bouton, une fenêtre d'alerte affiche la chaîne de caractère "Bonjour tout le monde".

Il existe plusieurs types d'événements, dont les principaux sont récapitulés dans le tableau suivant. Vous pouvez utiliser les événements sur la plupart des balises HTML.

Tableau 11.1 : *Les principaux événements JavaScript*

Nom de l'événement	Se produit lorsque...
`OnBlur`	La souris quitte la zone de la balise.
`OnClick`	L'utilisateur clique sur la zone délimitée par la balise.
`OnMouseOver`	La souris passe sur la zone délimitée par la balise.
`OnLoad`	La page est chargée par le navigateur.

Notez que l'événement `OnLoad` ne peut être défini que pour la balise `BODY`.

Distribuer un site

Lorsque vous avez fini de développer votre site et que vous en êtes fier, vous pouvez le mettre en ligne. Si vous disposez d'un accès à Internet, il est bien souvent couplé à une offre d'hébergement. Ayez recours à la documentation de votre fournisseur d'accès pour connaître les détails concernant la publication de contenus.

Vous avez maintenant une idée claire de ce qui compose une page web, qui n'est finalement rien d'autre qu'une suite de balises. Ce court

chapitre ne couvre pas l'intégralité du développement de site web et chaque section de ce chapitre pourrait faire l'objet d'un livre à elle seule. Le but ici était de vous présenter les trois principales technologies que sont HTML, JavaScript et les CSS, dans le cadre du développement de sites web à l'aide du Framework .NET. C'est ce que vous allez voir dans ce qui suit.

Chapitre 12

Les sites dynamiques

En ayant appris à développer un site HTML qui utilise JavaScript et des feuilles CSS, vous connaissez toutes les techniques de développement web utilisées jusqu'en l'an 2000. Depuis, le développement web ne s'est pas arrêté, il s'est simplement déplacé. Jusque-là, tout était fait du côté de l'ordinateur du client. Le schéma était alors le suivant :

1 Un client se connecte à un site web via une adresse du type **www.mapageweb.com**.
2 L'ordinateur qui héberge le site envoie à l'ordinateur client l'objet de sa requête, c'est-à-dire la page demandée.
3 Si la page contient du contenu JavaScript, le script est exécuté sur l'ordinateur client.

Cela convenait à tout le monde, ou presque.

L'un des principaux désavantages de cette méthode est que toute la page doit être envoyée au client, et ce à chaque requête. Cela implique que toute la page est disponible du côté du client. Que se passe-t-il si la page contient un mot de passe ? Il est en clair sur la page et tous ceux qui peuvent la lire le voient. D'un point de vue concurrentiel, cela signifie également que l'on peut sauvegarder toutes les pages sur un disque dur, modifier le site puis le remettre en ligne en s'appropriant le travail.

En bref, ce schéma de développement présentait quelques désavantages.

12.1. Le schéma client-serveur

Pour remédier à ces problèmes, les langages de développement côté serveur sont apparus. La principale différence est que la page web n'est plus stockée, telle qu'elle est affichée, sur l'ordinateur distant qui fait la requête au serveur. On utilise un langage qui, une fois interprété par le serveur, donne la page finale au client. Les étapes sont les suivantes :

1 Un ordinateur se connecte à un site web.
2 Le serveur qui héberge le site interprète la page demandée.
3 La page voulue est envoyée au client dans une forme compréhensible par son navigateur web.

Le principal avantage de cette méthode est la possibilité d'un véritable échange entre le client et le serveur.

De cette manière et en utilisant un langage complexe, vous pourrez proposer à un visiteur de véritables programmes sur le Web, à travers vos sites. Tous les sites modernes utilisent des technologies serveur.

Deux principales technologies se partagent le marché : d'un côté, les solutions dites "open source", avec les outils PHP, Apache et MySQL, et de l'autre, les solutions propriétaires représentées par Microsoft, avec les technologies ASP et SQL Server.

12.2. PHP

PHP est l'acronyme de "PHP Hypertext Preprocesseur", ce qui signifie en français "préprocesseur PHP". Il s'agit de générer une page web au format HTML pour la rendre lisible sur un navigateur.

Installer une solution PHP

Pour pouvoir développer des pages web en utilisant PHP, il faut installer un serveur sur votre machine. Il existe des packs préconfigurés. Leur utilisation est à la portée de tous. En outre, les utilisateurs forment une grande communauté et sont prêts à vous apporter leur aide pour peu que vous les sollicitiez gentiment.

La solution PHP tout en un la plus connue est certainement EasyPHP. En l'installant, vous disposez de tous les outils : un serveur web directement sur votre machine pour pouvoir tester vos pages, un moteur de base de données que vous pourrez utiliser avec vos pages et les outils d'administration graphiques.

Pour télécharger EasyPHP, rendez-vous à l'adresse **www.easyphp.org**. Téléchargez le programme d'installation, lancez-le et suivez les instructions.

Maintenant que votre machine est configurée pour exécuter des pages en PHP, lancez votre éditeur de texte et recopiez la page suivante :

```
<?php
Echo ("Ma première page Php");
?>
```

Enregistrez le fichier dans le dossier *www* du répertoire d'installation d'EasyPHP, en lui donnant l'extension *.php*.

Lancez maintenant votre navigateur Internet et rendre-vous à l'adresse **http://localhost/nomdevotrepage/php**. **localhost** est simplement un alias stipulant que la page est stockée sur votre serveur.



Les instructions

En PHP, les instructions se terminent par un point-virgule (`;`) en fin de ligne. Ne l'omettez pas, sous peine de rendre vos pages inutilisables.

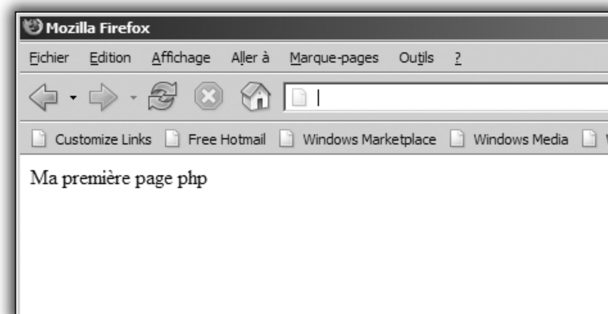


Figure 12.1 : *Votre première page*

Remarquez que le code PHP est placé à l'intérieur des balises `<?php` et `?>`.

L'utilité des technologies serveur est la réception et l'envoi de données. Vous allez voir comment procéder en recourant aux formulaires web.

Utiliser les formulaires web

Il est possible de déclarer des formulaires dans une page web en utilisant les balises `<form>` et `</form>`. Entre ces deux balises, vous pouvez placer des balises `input`, qui constitueront les éléments de votre formulaire.

1 Recopiez la page suivante dans un éditeur de texte :

```
<html>
<head>
    <title>Ma page de formulaire</title>
    <link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<form>
Nom:<input type=text name="nom"/>
Prénom:<input type=text name="prenom"/>
Âge:<input type=text name="age"/>
<input type=submit />
</form>
</body>
</html>
```

2 Sauvegardez la page au format HTML puis affichez-la dans votre navigateur.

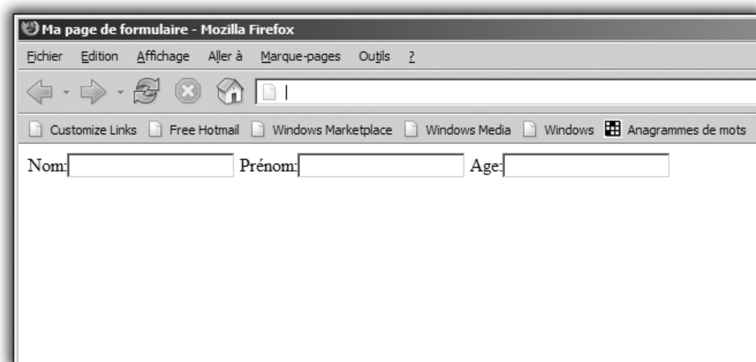


Figure 12.2 : *Le formulaire*

Vous disposez maintenant d'une page qui affiche trois zones de saisie de texte : *Nom*, *Prénom* et *Âge*.

Le problème est que, pour l'instant, vous ne pouvez rien en faire. Vous allez changer cela tout de suite.

3 Ouvrez un nouveau document texte et copiez le contenu suivant :

```
<?Php
$nom = $_POST["nom"];
```

```
$prenom= $_POST["prenom"];
$age=$_POST["age"]
echo "Vous vous appelez".$prenom." ".$nom." et vous
✂ avez ".$age." ans";
?>
```

Il s'agit de la page qui sera destinée au traitement des données du formulaire. Pour récupérer ces données, il convient de les stocker en utilisant des variables. Il faut pour cela les déclarer de cette façon : `$nomdelavariable=valeur`.

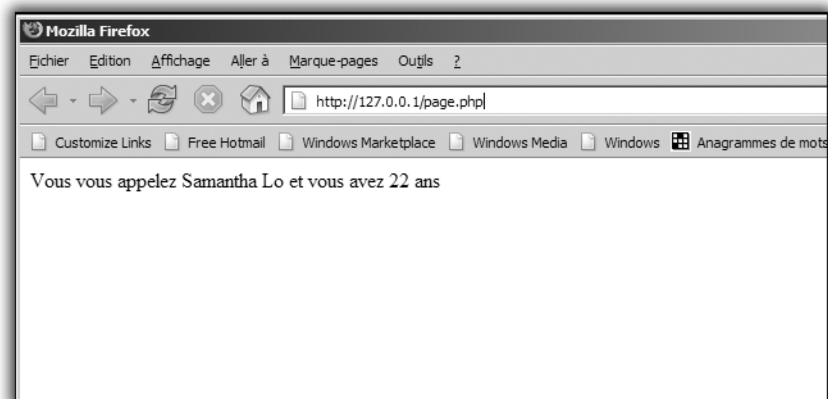


Figure 12.3 : *Le questionnaire traité*

Tout comme JavaScript et Visual Basic.Net, PHP est un langage faiblement typé. Il n'est donc pas nécessaire de préciser le type des variables lorsque vous les déclarez. La valeur que vous leur affectez ici est celle qui est récupérée depuis le formulaire. En PHP, lorsqu'un formulaire est utilisé, ses variables sont stockées différemment selon la méthode utilisée pour déclarer le formulaire. Il existe deux méthodes, `POST` et `GET`, sur lesquelles nous reviendrons. Vous utilisez ici la méthode `POST`.

Quelle que soit la méthode employée, la manière de stocker les variables du formulaire ne change pas. Elles sont enregistrées dans un tableau, chacune dans une case qui porte le nom donné à la balise `input` du formulaire. Le nom du tableau est toujours du type `$_POST[""]` ou `$_GET[]` selon la méthode utilisée.

Vous appelez ensuite la fonction `echo` de PHP, qui permet d'afficher du texte sur une page. Il suffit de lui passer en paramètre les chaînes et les variables dont vous souhaitez afficher le contenu.

4 Il faut maintenant faire le lien entre le formulaire et son traitement. Modifiez la page d'origine conformément au modèle suivant :

```
<html>
<head>
    <title>Ma page de formulaire</title>
    <link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<form method="POST" action="traitement.php">
Nom:<input type=text name="nom"/>
Prénom:<input type=text name="prenom"/>
Age:<input type=text name="age"/>
<input type=submit />
</form>
</body>
</html>
```

Il subsiste une ligne non encore décrite. Il s'agit de :

```
<input type=submit />
```

Elle a pour but d'ajouter à la page de formulaire un bouton de validation qui valide les données et les envoie à la page de traitement spécifiée dans l'attribut `action` de la balise ouvrante du formulaire. Copiez ces deux pages dans le dossier *www* du répertoire d'installation d'EasyPHP.

Les deux méthodes

L'attribut `method` de la balise ouvrante du formulaire permet de spécifier si l'on souhaite utiliser la méthode `POST` ou `GET` pour transmettre les données. `GET` transmet les données dans l'adresse de la page. C'est le cas par exemple des données transmises à un moteur de recherche comme Google. Si vous allez sur le site et que vous faites une recherche, l'adresse de la page de résultat contiendra cette recherche.

En d'autres termes, une fois que vous cliquerez sur le bouton de validation du formulaire, la page de traitement sera chargée ; son adresse

contiendra le nom du paramètre et sa valeur, sous la forme **www.mapagedetraitement.com?nomduparamètre=valeur**. Vous pouvez prendre l’exemple suivant pour effectuer une recherche sur Google à partir de votre site :

```
<html>
<head runat="server">
    <title>Ma page de formulaire</title>
    <link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<form method=GET action=http://www.google.fr>
Recherche:<input type=text name=q/>
<input type=submit />
</form>
</body>
</html>
```

Dans la partie `body` de la page, vous déclarez un formulaire. La méthode est `GET` et le traitement est reporté à la page **www.google.fr**. Dans ce formulaire, vous ajoutez un champ de saisie de texte appelé `q`, puis un bouton de validation.

Chargez la page dans un navigateur, entrez un ou plusieurs mots dans le champ de texte et appuyez sur le bouton de validation. Votre navigateur va charger la page de traitement du formulaire, en ajoutant les données saisies dans le champ de recherche. L’adresse chargée devient alors **www.google.fr?q=mot+recherché**.



Récupération des données

Bien que les données de formulaire soient passées dans la barre d’adresse, il n’est pas possible de les récupérer pour les traiter autrement qu’en lisant le tableau de données `GET[]` associé au formulaire.

La méthode `POST` transmet les données dans l’en-tête de la page, c’est-à-dire de manière complètement transparente pour l’utilisateur. L’avantage principal est de ne pas exposer les données sensibles à n’importe qui. Si par exemple vous demandez une identification par mot de passe sur votre site en faisant appel à la méthode `GET`, le mot de passe apparaît en clair dans la barre d’adresse du navigateur, ce qui est à proscrire dans le cadre d’une application sécurisée.

Les instructions de contrôle

PHP propose des méthodes de contrôle, au même titre que Visual Basic .NET. Toutes les boucles et structures classiques, comme `If… Then… Else` ou `Switch`, sont utilisables. Leur syntaxe en PHP, comparativement à Visual Basic .NET, est légèrement différente. Voici un récapitulatif de ces structures de contrôle en PHP.

Si… Alors… Sinon

La boucle de type `If` se décrit de cette manière :

```
<html>
<head runat="server">
<title>Ma page de If</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<?php
$variable = 150
if ($variable<250){
 echo("La variable est plus petite que 250");
}
else{
 echo("La variable est plus grande que 250");
}
?>
</body>
</html>
```

Notez bien l'utilisation d'accolades pour délimiter les blocs d'instructions. Si vous avez plusieurs tests successifs, vous pouvez utilisez la structure `If… ElseIf… Else`, qui permet d'imbriquer plusieurs tests, de la manière suivante :

```
<html>
<head runat="server">
<title>Ma page de If</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<?php
$variable = 150
if ($variable<250){
 echo("La variable est plus petite que 250")
}
elseif($variable>250){
 echo("La variable est plus grande que 250")
}
```

```
else{
  echo("La valeur de la variable n’a pas été testée")
}
?>
</body>
</html>
```

Il existe une autre façon d’écrire le contrôle `If`, sans les accolades :

```
<html>
<head runat="server">
<title>Ma page de If</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<?php
if(votre test):
  Instructions;
else:
  Instructions;
?>
</body>
</html>
```

Attention car cette écriture n’est pas compatible avec toutes les versions de PHP.

Les boucles

Les boucles itératives permettent de répéter une ou plusieurs instructions.

While

La boucle `While`, également proposée par Visual Basic .NET, a la syntaxe suivante en PHP :

```
While(votrecondition){
//Vos instructions
}
```



Commentaires

Les doubles barres obliques correspondent à la mise en commentaire d’une ligne.

La boucle While permet de répéter les instructions tant que le test conditionnel est vrai. À cause de cela, si vous ne pensez pas à mettre un cas d'arrêt, c'est-à-dire un modificateur de la condition de test, la boucle ne s'arrêtera jamais. Heureusement, la plupart des navigateurs récents arrêtent le programme automatiquement au bout de 30 secondes.

For

La boucle For permet, contrairement à la boucle While, de préciser le nombre d'itérations à partir d'une condition initiale et l'arrêt de la boucle avec une condition d'arrêt.

Elle est utile lorsque vous raisonnez en termes de nombre d'itérations plutôt qu'en termes de condition.

La syntaxe de la boucle For en PHP est la suivante :

```
$i;
$j=2;
For($i=0;$i<10;$i++){
   Echo $i*$j;
}
```

Vous affichez ici la table de multiplication par 2. Sur les deux premières lignes, vous initialisez deux variables i et j. La première sert de compteur, la seconde de variable pour la table de multiplication.

Dans la boucle for, vous précisez l'initialisation, ici la valeur de base de i, ensuite le cas d'arrêt, à savoir une sortie de la boucle d'exécution quand i est égal à 9 (il faut donc tester si la valeur de i est strictement inférieure à 10), et enfin le pas, ce qui représente la modification appliquée à la valeur de i à chaque itération de la boucle.

Switch

La structure Switch permet de remplacer avantageusement un grand nombre de tests conditionnels :

```
<html>
<head runat="server">
<title>Ma page de If</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<?php
if ($i == 0) {
    print "i égale 0";
```

```
}
if ($i == 1) {
    print "i égale 1";
}
if ($i == 2) {
    print "i égale 2";
}
switch ($i) {
    case 0:
        print "i égale 0";
        break;
    case 1:
        print "i égale 1";
        break;
    case 2:
        print "i égale 2";
        break;
}
?>
</body>
</html>
```

Ces deux blocs de code sont équivalents. Le `Switch` permet de clarifier la syntaxe qui peut être brouillonne si vous multipliez les tests.



Instruction Break

L'instruction `break` des éléments `case` de la structure précédente permet d'arrêter le traitement des instructions. Si vous l'omettez, toutes les instructions de tous les `case` seront exécutées.

Les fonctions

Il est également possible de déclarer des fonctions en PHP, avec tous les avantages que cela amène. Pour écrire des fonctions en PHP, il suffit d'utiliser le mot-clé `Function` de la manière suivante :

```
<html>
<head runat="server">
<title>Ma page de fonctions</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<?php
Function nomdelafonction($argument1,$argument2){
Echo $argument1+$argument2;
?>
```

```
</body>
</html>
```

Ici, la fonction va effectuer un affichage sur la page de deux arguments qu'elle reçoit en paramètre.



Tests

Ce genre de fonctions est utile pour effectuer des tests sur un projet afin de trouver d'où viennent les problèmes.

Vous pouvez maintenant structurer correctement les fichiers de votre site en organisant au mieux votre code. Pour bien réutiliser du code, il convient de le placer dans un fichier à part. Imaginez une fonction qui affiche le formulaire de recherche sur Google sur chacune des pages de votre site :

```
<html>
<head runat="server">
<title>Ma page de fonctions</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>

<?
Function QuestionGoogle(){
<form method=GET action=http://www.google.fr>
Recherche:<input type=text name=q/>
<input type=submit />
</form>
}
?>
</body>
</html>
```

Vous allez la placer dans un fichier à part : *fonctions1.php*. Pour réutiliser le contenu de *fonctions1.php* dans un autre fichier, il suffit d'ajouter la ligne suivante :

```
<?include "fonctions1.php"?>
```

En ajoutant cette ligne en haut du fichier courant, vous pourrez utiliser tout le code placé à l'intérieur de *fonctions1.php* comme si vous l'aviez copié-collé.

12.3. PHP côté serveur

Vous savez à présent développer des scripts PHP qui permettent un retour d'informations au client, c'est-à-dire à l'ordinateur qui se connecte au site web.

Il y a deux manières d'interagir avec le client :

- soit en utilisant des fichiers cookies, qui enregistrent des informations sur l'ordinateur du client ;
- soit en utilisant des sessions, qui stockent temporairement des informations dans la mémoire de l'ordinateur qui héberge le site web.

Le cookie est un fichier texte qui peut contenir la valeur que vous voulez. Vous pouvez faire en sorte que le fichier cookie expire lorsque l'utilisateur quitte votre site. Mais il est courant de l'utiliser pour enregistrer par exemple la date de la dernière visite de l'internaute. Si vous ne prévoyez pas un délai assez large entre deux visites, vous perdrez l'information.

Enregistrer un cookie

Pour enregistrer un cookie sur l'ordinateur du client, utilisez le code suivant :

```
<html>
<head runat="server">
<title>Ma page de fonctions</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>

<?php
$valeur = "Valeur de test";

setcookie("MonCookie", $valeur);
?>
</body>
</html>
```

La fonction `setcookie()` définit le fichier à écrire sur l'ordinateur client ou plutôt le contenu du fichier.



Plusieurs valeurs pour un cookie

Il n'est permis d'écrire qu'une seule valeur par fichier cookie. Aussi, si vous comptez stocker plusieurs informations, vous devez utiliser un séparateur de texte, par exemple la barre oblique ou un point-virgule.

Selon la manière dont l'utilisateur a configuré son navigateur Internet, le fichier est maintenant stocké quelque part sur le disque dur de l'ordinateur client. Pour en lire le contenu, il faudra utiliser un code ressemblant à celui-ci :

```
<html>
<head runat="server">
<title>Ma page de fonctions</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>

<?php
echo $_COOKIE["MonCookie"];
?>
</body>
</html>
```

La lecture des cookies se fait par un tableau les contenant tous. En inscrivant le nom de votre cookie (le premier paramètre de la fonction `setcookie`) en index du tableau, vous pourrez en récupérer la valeur et l'utiliser directement. Dans l'exemple précédent, il est question de l'afficher, mais vous pouvez le stocker dans une variable, comme suit :

```
<html>
<head runat="server">
<title>Ma page de fonctions</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>

<?php
$Variable = $_COOKIE["MonCookie"];
?>
</body>
</html>
```

Les cookies sont des systèmes de stockage d'informations utilisés plutôt sur le long terme. Pour le court terme, il est d'usage de recourir au mécanisme de sessions.

Organiser des sessions

Les sessions sont un peu l'équivalent des variables globales de Visual Basic .NET. En effet, lorsque vous déclarez une variable en PHP, elle n'est visible qu'à l'intérieur de la page dans laquelle se trouve la définition. Si un utilisateur change de page, la variable est détruite. Pour éviter cela, il est possible de définir une session, dotée d'un numéro qui sera unique, vous permettant d'une part d'identifier les utilisateurs, d'autre part de garantir la sécurité des données leur appartenant.

Le mécanisme de sessions doit être démarré au début de chaque page à l'aide de la commande :

```
<?
Session_start();
?>
```

Cette ligne induit deux comportements différents :

- Soit l'utilisateur vient d'une page où les sessions existaient déjà, il dispose donc d'un numéro unique de session qui est réactivé sur cette page.
- L'utilisateur vient d'une page où les sessions n'étaient pas initialisées et un numéro unique est initialisé dans les informations de navigation de l'internaute sur le serveur.

Activer une session

Maintenant que la session est activée, il suffit de s'en servir pour enregistrer des variables de session de la manière suivante :

```
<?php
Session_start();
$Variable = "valeur";
Session_register("Variable");
?>

<html>
<head runat="server">
<title>Ma page de sessions</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>

</body>
</html>
```

Dans cet exemple de page, vous déclarez une variable contenant la chaîne "valeur", qui sera ensuite intégrée à la collection des variables de session. Le paramètre de Session_register() est le nom d'une variable et n'est pas précédé du caractère $.

Attention : les initialisations de session doivent figurer en tout début de fichier. En effet, vous risquez d'avoir des erreurs dans l'exécution du script de votre page si le démarrage des sessions se fait au milieu de la page.

Utiliser les variables de session

Maintenant qu'une variable de session est enregistrée, il faut l'utiliser. Il suffit pour cela d'avoir recours au tableau de valeurs $_SESSION[nom de la valeur]. Le code suivant permet d'afficher sur une deuxième page le contenu de la variable enregistrée précédemment :

```
<?php
Session_start();
?>

<html>
<head runat="server">
<title>Ma page de sessions</title>
<link rel=stylesheet href=StyleSheet.css />
</head>
<body>
<?
Echo $_SESSION[Variable]
?>
</body>
</html>
```

Lorsque vous chargez cette page, le contenu de la variable enregistrée précédemment s'affiche.

Chapitre 13

Web dynamique et .NET : ASP .NET

Jusqu'à maintenant, vous avez vu la plupart des techniques employées en programmation avec Visual Basic .NET ainsi que les principes fondamentaux du développement de sites web dynamiques avec PHP. Imaginez une jointure entre les deux mondes. Supposez un instant que vous puissiez utiliser la facilité de développement de .NET dans une application web. Si vous avez déjà une légère expérience de ce que peut être le développement d'un site complet en PHP, l'idée doit vous plaire. Si vous êtes encore débutant, imaginez simplement que vous allez construire des sites web complexes aussi facilement que vous avez créé des applications Windows complètes dans les chapitres précédents.

Ce monde où le meilleur des deux principes de développement se rejoint, c'est ASP .NET. ASP .NET n'est pas un langage, c'est un ensemble de technologies. ASP est l'acronyme d'"Active Server Pages". Il s'agit d'un ensemble d'outils prêts à l'emploi permettant de tout gérer côté serveur. En d'autres termes, toute la gestion des formulaires, des sessions, etc. est déjà implémentée. En développant ASP .NET, Microsoft a réfléchi pour vous. Vous n'avez plus qu'à utiliser ces outils mis à votre disposition.

13.1. L'éditeur, le langage

Nous avons dit qu'ASP .NET n'est pas un langage. Cela implique que pour l'utiliser, il faut en connaître un. Heureusement c'est le cas, puisque les pages peuvent être écrites en Visual Basic .NET.

L'éditeur, par contre, change. Il faut vous procurer Visual Web Developer Express, disponible sur le site de Microsoft. Comme pour PHP, il faut un serveur web sur votre machine. Cependant, Visual Web Developer Express contient un serveur qui sera automatiquement associé au lancement de vos sites.

Créer un projet

Comme pour les projets Visual Basic .NET, il faut passer par le menu **Fichier/Nouveau projet** pour commencer le développement d'un nouveau site. Vous retrouvez alors l'Explorateur de solutions, la fenêtre des propriétés, la boîte à outils, etc. Tous les éléments qui vous ont aidé à créer vos applications sont de nouveau disponibles.

La page default.aspx

Lors de la création d'un projet, une page par défaut est ajoutée à votre solution. Il s'agit de la page par défaut de votre application. Si vous appuyez sur F5 dès maintenant, vous allez charger cette page dans votre navigateur et elle sera blanche. Commencez par lui ajouter du contenu :

1 Cliquez du bouton droit sur la page puis sélectionnez **Vue Design**.
2 Ajoutez des contrôles depuis la boîte à outils, comme vous le feriez dans un projet classique.
3 Appuyez sur F5 pour tester l'application. Cela marche.

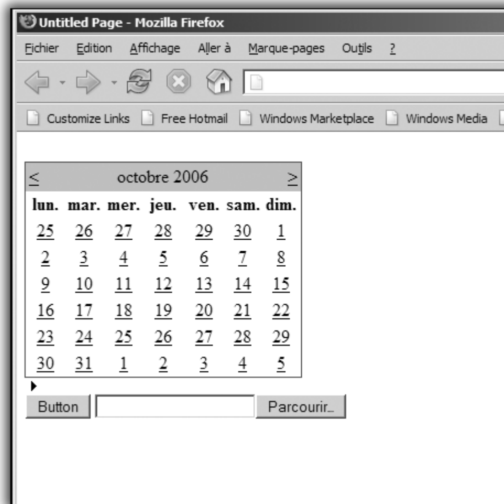


Figure 13.1 : *Un exemple de site avec un contrôle calendrier*

Vous pouvez ajouter d'autres pages de type HTML, *.asp*, *.aspx*, etc. en cliquant du bouton droit sur le nom de votre site dans l'Explorateur de solutions et en choisissant **Ajouter un nouvel élément**.

Voyons maintenant comment ajouter un peu d'interactivité à ce nouveau site.

13.2. Les contrôles web

Pour développer des applications Windows, vous avez utilisé des contrôles formulaires. Cette fois, vous aurez recours à des contrôles

web. Ce sont bien souvent des adaptations de ce que l'on trouve généralement sur les sites Internet. Les développeurs ont ainsi l'outil ad hoc prêt à l'emploi, ce qui leur facilite la tâche.

Par exemple, tous les sites web ont besoin de boutons pour ajouter de l'interaction avec le navigateur. Ainsi, dans la boîte à outils de Visual Web Developer Express, vous trouverez un contrôle *Button* et un contrôle *ImageButton*. Le premier est un bouton classique qui se comportera comme celui des applications Windows. Le second a une propriété *Image* qui sera son image de fond. Cette image sera "cliquable" pour simuler un clic de bouton.

En faisant glisser un contrôle sur la page *default.aspx*, vous constatez que vous ne pouvez pas le déplacer comme sur un formulaire Windows. C'est normal. Pour placer vos éléments sur la page, vous devez utiliser une feuille de style CSS.

Voici un exemple de petit site permettant d'afficher des images à la manière d'un blog.

1 Ouvrez un nouveau projet de type site web ASP .NET.
2 Ajoutez un contrôle *FileUpload*.
3 Ajoutez un contrôle *Button*.

À présent, appuyez sur F5.

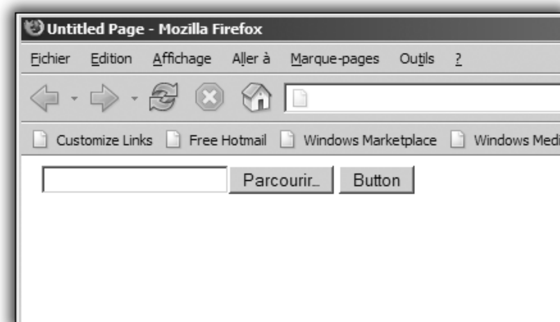


Figure 13.2 : *Le contrôle FileUpload*

Le contrôle *FileUpload* permet de gérer l'envoi de fichiers vers le serveur web. Cependant, il faut préciser quand et où sauvegarder le fichier. Le "quand" se précise sur un événement de la page, par exemple le clic d'un bouton.

Dans le designer de pages web, double-cliquez sur le bouton pour vous placer dans l'événement "clic", puis entrez l'instruction suivante :

```
FileUpload1.SaveAs("c:\image.jpg")
```

Cette instruction précise où et sous quel nom le fichier sélectionné peut être sauvegardé.



Chemin de sauvegarde

Le contrôle *FileUpload* n'accepte que les chemins complets, et non les chemins dits "abrégés" de type "*./repertoire/repertoire2*".

Vous avez maintenant envoyé votre image au serveur. Encore faut-il l'afficher sur la page. Pour cela, vous allez ajouter un contrôle *Image*. Ensuite, ajoutez la ligne suivante au code de l'événement "clic" du bouton :

```
Image1.ImageUrl = "c:\image.jpg"
```

Cette ligne affecte une image au contrôle.

Relancez la page avec F5. Sélectionnez une image puis cliquez sur le bouton. L'image s'affiche sur la page.

Si vous testez cette méthode sur plusieurs navigateurs, vous risquez d'avoir des résultats différents. Cela vient du fait que les navigateurs ont une gestion différente des chemins, c'est-à-dire des adresses des fichiers sur le disque dur.

Vous êtes maintenant en mesure d'ajouter à votre site Internet tous les contrôles disponibles et de les utiliser. Vous pouvez reprendre le système de connexion à une base de données, utiliser les événements, etc.

Chapitre 14

Annexes

14.1. Glossaire de programmation

Voici les termes qui reviennent le plus souvent en programmation :

Base de données

Système de classement et de stockage d'informations.

Bibliothèque

Ensembles de classes et de fonctions destinées à être réutilisées.

Bogue

Problème de comportement d'un programme.

Caractère

Symbolise une variable destinée à stocker une lettre ou un symbole.

Chaîne

Tableau de caractères.

Classe

Ensemble de méthodes et de propriétés.

Code source

Fichier contenant les instructions à compiler pour obtenir un programme.

Compilateur

Programme destiné à transformer un code source en code exécutable par un ordinateur.

Compilation

Appel à un compilateur.

Contrôle

Dans le contexte de la programmation d'applications Windows, il s'agit d'une classe destinée à faciliter l'interaction avec l'utilisateur.

Cookies

Fichiers permettant de stocker des informations lorsqu'un utilisateur se connecte à un site web.

Déboguer

Corriger les failles d'un programme.

Éditeur

Programme utilisé pour l'édition de code source.

Entier

Symbolise une variable destinée à stocker une valeur numérique entière.

Fichier

Ensemble de données destiné au stockage.

Flottant

Symbolise une variable destinée à stocker une valeur numérique décimale.

Fonction

Ensemble d'instructions qui forment une portion de code réutilisable. À la différence d'une méthode, une fonction renvoie une valeur.

Formulaire

Dans le cadre de la programmation d'applications Windows, il s'agit d'une fenêtre destinée à assurer la communication d'informations entre le programme et l'utilisateur.

IDE

Environnement de développement. Il s'agit de programmes destinés à regrouper tous les outils nécessaires à la programmation, comme Visual Basic Express.

Index

Valeur numérique utilisée pour le positionnement dans un tableau.

Langage de programmation

Ensemble de termes et d'instructions.

Méthode

Ensemble d'instructions qui forment une portion de code réutilisable.

Navigateur

Permet d'afficher les pages web.

Objet

Instance d'une classe.

Projet

Ensemble des codes sources d'un programme.

Ramasse-miettes (Garbage Collector)

Programme tournant en tache de fond et s'assurant de la bonne gestion de la mémoire.

Serveur web

Ordinateur qui stocke les pages d'un site Internet.

Solution

Ensemble de projets.

Système d'exploitation

Plate-forme destinée à l'exécution du programme.

Variable

Emplacement mémoire réservé au stockage d'une donnée d'un type défini.

14.2. Raccourcis clavier

Voici une liste non exhaustive des raccourcis clavier utiles dans Visual Basic Express.

Tableau 14.1 : *Liste de raccourcis clavier utiles*

Raccourci	Action
Ctrl+N	Créer un nouveau projet
Ctrl+O	Ouvrir un projet existant
Ctrl+S	Sauvegarder le fichier en cours
Ctrl+Maj+S	Sauvegarder tous les fichiers en cours
Ctrl+H	Rechercher et remplacer une chaîne
Ctrl+Alt+L	Afficher l'Explorateur de solutions
Ctrl+Alt+X	Afficher la boîte à outils
Ctrl+Alt+S	Afficher l'Explorateur de bases de données
Maj+F4	Afficher les propriétés d'un contrôle
F1	Afficher l'aide de Visual Studio

14.3. Sites web

Il existe sur Internet tout un réseau de sites avec des forums sur lesquels vous serez assuré d'obtenir des réponses rapides et fiables à vos questions si vous sollicitez les participants poliment.

Microsoft Developer Network

http://msdn2.microsoft.com/fr-fr/

La MSDN est la plus grande base de connaissances concernant le développement avec les technologies .NET et le développement Microsoft.

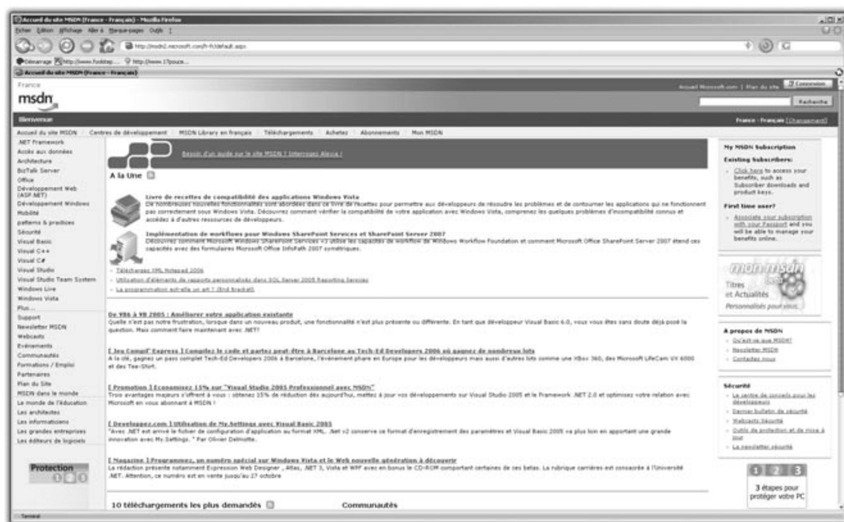


Figure 14.1 : *http://msdn2.microsoft.com/fr-fr/*

Le laboratoire .NET

www.labo-dotnet.com

Le site du laboratoire des technologies .NET de SUPINFO. Vous pouvez y télécharger des cours, travaux pratiques, et utiliser le forum pour poser des questions.



Figure 14.2 : *www.labo-dotnet.com*

Le réseau Codes-sources

www.codes-sources.com

Le réseau Codes-sources est un ensemble de sites dédiés à différents langages. Vous y trouverez des exemples illustrés pour réaliser bon nombre d'opérations de programmation.



Figure 14.3 : *www.codes-sources.com*

Le club développez.com

www.developpez.com/

Le site developpez.com met à votre disposition des articles, des exemples et un forum très actif pour vous permettre de progresser en programmation.

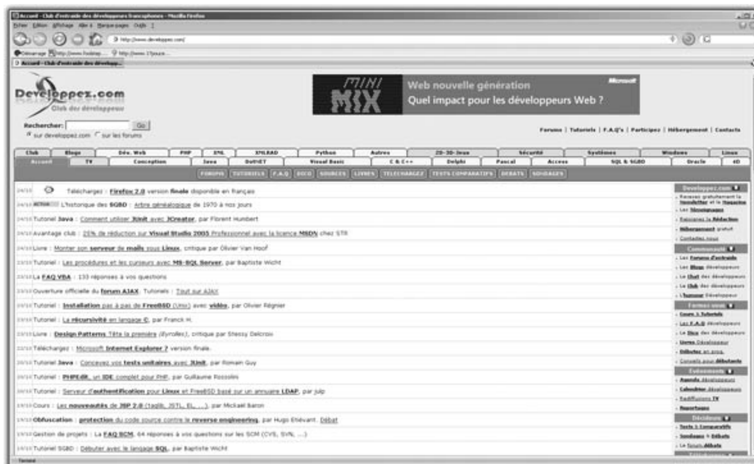


Figure 14.4 : *www.developpez.com/*

14.4. Blogs

La plupart des développeurs professionnels possèdent un blog sur lequel ils exposent leurs soucis techniques et les solutions qu'ils ont trouvées. Voici quelques adresses intéressantes. Cette liste n'est pas complète et n'a pas la prétention de l'être. Lorsque vous restez bloqué sur un problème de développement, saisissez votre problème sur un moteur de recherche. Quelqu'un a probablement eu le même problème avant vous et l'a peut-être déjà résolu.

Patrice Lamarche

http://blogs.developpeur.org/patrice/

Patrice est un ancien du réseau Codes-sources. Son blog fournit des informations sur le développement .NET en général.



Figure 14.5 : *http://blogs.developpeur.org/patrice/*

Simon Ferquel

http://blogs.labo-dotnet.com/simon/

Le futur du développement. Simon est un passionné de recherche et développement et son blog est une véritable vitrine technologique. Attention : le niveau demandé pour la compréhension du code est élevé.

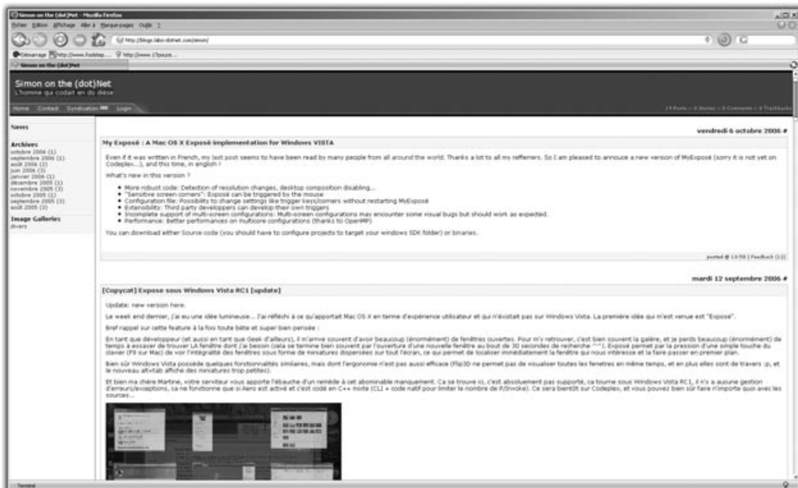


Figure 14.6 : *http://blogs.labo-dotnet.com/simon/*

Thomas Lebrun

http://morpheus.developpez.com/

Thomas est membre de l'équipe de rédaction de développez.com et spécialiste de l'interaction entre le développement .NET et la suite Office.



Figure 14.7 : *http://morpheus.developpez.com/*

Les blogs MSDN

Les développeurs en charge de maintenir la bibliothèque ont chacun leurs blogs, disponibles sur *http://blogs.msdn.com*.

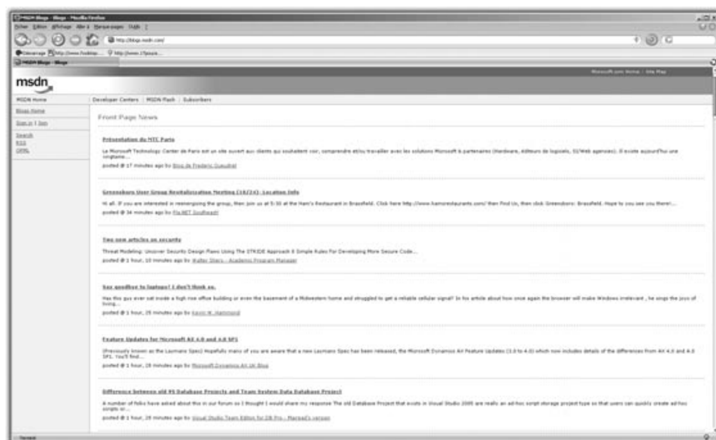


Figure 14.8 : *http://blogs.msdn.com*

14.5. Comparatif des langages

En lisant ce livre, vous avez appris à manipuler le langage Visual Basic .NET. Il existe de nombreux autres langages, chacun ayant leurs avantages et leurs inconvénients. Vous trouverez ici un résumé des principaux langages utilisés aujourd'hui.

C

C est un langage qui date de la fin des années 70. Il est souvent enseigné à l'université car, du fait de son ancienneté, il existe beaucoup de développeurs expérimentés capables de transmettre leur savoir. Le principal avantage de C est de permettre une grande interaction avec la machine au niveau physique. En effet, il n'y a aucun ramasse-miettes et il faut gérer soi-même l'espace mémoire du programme. C'est un peu comme une voiture de sport sans assistance au freinage. Il est possible d'aller très vite et d'arriver en tête de la course, mais à la moindre erreur, vous payez plein tarif. C est très utilisé en environnement Linux, ce système étant entièrement développé en C.

Exemple :

```
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  puts("Hello World!");
  return EXIT_SUCCESS;
}
```

C++

Apparu quelques années après C, C++ en reprend les idées fondamentales et ajoute les concepts de la programmation orientée objet. C++ n'a toujours pas de ramasse-miettes et est aujourd'hui couramment utilisé pour les applications scientifiques demandant une forte optimisation, et pour les jeux. Ce langage est très performant car proche de la machine, mais attention aux erreurs.

Exemple :

```
#include <iostream.h>

main()
```

```
{
  cout << "Hello World!" << endl;
  return 0;
}
```

C#

Prononcez "C Sharp". C# est un langage inventé par Microsoft pour promouvoir les technologies .NET. Son principal avantage est une syntaxe proche de Java et surtout un environnement managé permettant une grande souplesse de développement. En ajoutant toutes les fonctionnalités du Framework .NET, vous obtenez un langage permettant de développer rapidement des applications professionnelles.

```
using System;

namespace Exemple
{
  class Class1
  {
    static void Main(string[] args)
    {

        Console.WriteLine("Hello, World");
    }
  }
}
```

Java

Java est le grand concurrent de C#, développé par Sun. Java offre à peu près les mêmes avantages : un environnement managé, des bibliothèques de fonctions, le concepts de la programmation orientée objet. Au moment de la rédaction de ce livre, Java représente environ 55 % des développements logiciels en entreprise. Si vous cherchez un nouveau langage pour compléter votre savoir, Java peut être un bon choix.

```
public static void copyFile(File src, File dest) throws
✂ IOException
{FileInputStream fis = new FileInputStream(src);
  FileOutputStream fos = new FileOutputStream(dest);

  java.nio.channels.FileChannel channelSrc   = fis
  ✂ .getChannel();
  java.nio.channels.FileChannel channelDest = fos
  ✂ .getChannel();
```

```
  channelSrc.transferTo(0, channelSrc.size() ,
  ✂ channelDest);

  fis.close();
  fos.close();
}
```

PHP

PHP est un langage à la mode dans le monde du développement de sites Internet. Il permet une interaction poussée avec l'utilisateur et sa facilité d'accès en fait un bon langage pour le développement d'applications web. Il est aujourd'hui devenu incontournable dans le monde des intranets d'entreprise.

```
<?php
  // Hello World en PHP
  echo 'Hello World!';
?>
```

ASP

Plus qu'un langage, ASP (Active Server Pages) est une technologie développée par Microsoft pour concurrencer PHP. Proposant une interaction forte avec l'environnement Windows, son évolution ASP .NET avec le Framework .NET constitue un outil puissant dans le développement de sites Internet dynamiques.

```
<%@ language="vbscript" %>
<html><body>
<%
Response.write "Hello World!"
%>
</body></html>
```

HTML

HTML est un langage destiné à la présentation des sites web. Il n'est pas compilé, mais interprété par le navigateur au moment du chargement de la page. Très facile d'accès, c'est le langage à connaître pour la réalisation de sites Internet.

```
<HTML>
<HEAD>
<TITLE>Hello World!</TITLE>
```

```
</HEAD>
<BODY>
Hello World!
</BODY>
</HTML>
```

14.6. Mots clés du langage Visual Basic .NET

Voici une liste non exhaustive des mots clés du langage, rappelez vous que vous ne pourrez pas les utiliser pour un autre usage que celui prévu par Visual Basic.

Mot clé	Utilisation
`As`	Spécifie un type de variable
`Boolean`	Type de variable Booléen
`Date`	Type de variable Date
`Decimal`	Type de variable pour les nombres à virgule
`Dim`	Mot clé de déclaration de variable
`Do`	Définit le départ d'une boucle `Do...While`
`Double`	Déclare un grand nombre
`Each`	Partie de la boucle For Each
`Else`	Test conditionnel contraire
`ElseIf`	Mot clé d'imbrication de tests
`End`	Marque la fin d'une procédure ou d'une fonction
`False`	Booléen faux
`Finally`	Mot clé de gestion d'erreurs
`For`	Structure de contrôle
`Function`	Mot clé de déclaration de fonction
`If`	Test conditionnel
`In`	Utilisé dans la boucle `For…Each` pour spécifier une collection
`Inherits`	Permet l'héritage

Interface	Ensemble de propriétés et méthode que doit implémenter une classe
`Is`	Test de type pour une variable
`Me`	Mot clé permettant l'accès aux ressources de la classe
`New`	Appel du constructeur de la classe
`Next`	Passe à l'itération suivante d'une boucle `For`
`Not`	Négation
`Object`	Type de base
`Private`	Modifie la portée d'une variable
`Protected`	Modifie la portée d'une variable
`Public`	Modifie la portée d'une variable
`Return`	Spécifie la valeur de retour d'une fonction
`Short`	Type de variable pour les petits nombres
`Static`	Modifie l'accès aux données d'une classe
`String`	Type de donnée chaîne de caractères
`Sub`	Spécifie le début d'une procédure
`Then`	Complément du type conditionnel
`Throw`	Permet la gestion d'exception
`True`	Booléen vrai
`Try`	Permet le test d'une instruction pour en gérer les exceptions
`Until`	Boucle s'exécutant jusqu'à ce qu'une condition soit vraie
`While`	Boucle s'exécutant tant qu'une condition est vraie
`Xor`	Ou exclusif

14.7. Les Balises HTML

Voici une liste des balises HTML les plus utilisées:

Balise	Utilisation
A	Définit un lien ou une ancre
BODY	Corps de la page
BR	Effectue un retour à la ligne
BUTTON	Insère un bouton
CAPTION	Permet de légender un tableau
FORM	Définit un formulaire web
FRAMESET	Division de la fenêtre
FRAME	Cadre de division
Hx	Avec x entre 1 et 6, permet de définir un titre
HEAD	Délimite l'entête de la page
HTML	Définit le début et la fin d'une page web
IMG	Permet d'insérer une image
INPUT	Permet la saisie de données dans un formulaire
LI	Objet d'une liste
LINK	Lien indépendant
OPTION	Option sélectionnable dans un formulaire
P	Délimite un paragraphe
SCRIPT	Délimite un script
SELECT	Définit le début et la fin d'un groupe d'options sélectionnables
TABLE	Définit un tableau
TD	Cellule de données d'un tableau
TEXTAREA	Définit une zone de texte
TH	Définit l'entête d'un tableau
TITLE	Titre de la page
TR	Ligne d'un tableau
UL	Liste non numérotée

14.8. Récapitulatif des projets

Voici l'intégralité du code nécessaire à réaliser les projets du livre.

L'album photo

Code à exécuter au chargement de l'application:

```
If FolderBrowserDialog1.ShowDialog = Windows.Forms
✂ .DialogResult.OK Then
For Each fichier As String In System.IO.Directory
✂ .GetFiles(FolderBrowserDialog1.SelectedPath)
If System.IO.Path.GetExtension(fichier) = ".jpg" Then
Dim MonImage As ListViewItem = ListView1.Items
✂ .Add(fichier)
MonImage.Name = fichier
MonImage.Text = System.IO.Path
✂ .GetFileNameWithoutExtension(fichier)
ImageList1.Images.Add(Image.FromFile(fichier))
MonImage.ImageIndex = MonImage.Index
End If
Next
End If
```

Instructions à exécuter lors du clic sur un bouton :

```
If FolderBrowserDialog1.ShowDialog = _
   Windows.Forms.DialogResult.OK Then
      ListView1.Items.Clear()
      ImageList1.Images.Clear()
      For Each fichier As String In _
       System.IO.Directory.GetFiles( _
       FolderBrowserDialog1.SelectedPath)
      If System.IO.Path.GetExtension(fichier) _
        = ".jpg" Then
      Dim MonImage As ListViewItem = _
      ListView1.Items.Add(fichier)
        MonImage.Name = fichier
        MonImage.Text = System.IO.Path
        ✂ .GetFileNameWithoutExtension(fichier)
        ImageList1.Images.Add(Image.FromFile(fichier))
        MonImage.ImageIndex = MonImage.Index
      End If
      Next
End If
```

Instructions à exécuter lors du clic sur le bouton de rotation de l'image :

```
Dim img As Image = Form1.PictureBox1.Image
   Dim g As Graphics = e.Graphics

   Dim M As New Matrix
   M.RotateAt(90, New PointF(Width / 2, Height / 2))
   g.Transform = M
   g.DrawImage(img, 0, 0)
```



Importation

N'oubliez pas d'importer le fichier *System.drawing.Drawing2D*.

```
Imports System.Drawing.Drawing2D
```

Affichez la fenêtre de l'image retournée :

```
Dim form2 As Rotation90
form2 = New Rotation90
form2.Show()
```

Lecteur Multimédia

Ajoutez à votre projet la référence *Microsoft.DirectX.AudioVideoPlayback* puis un bouton et une boîte de dialogue pour la sélection d'un fichier. Pour l'événement clic d'un bouton, ajoutez le code suivant :

```
If OpenFileDialog1.ShowDialog = Windows.Forms.DialogResult
✂ .OK Then
   Dim PlayAudio As New Microsoft.DirectX
   ✂ .AudioVideoPlayback.Audio(OpenFileDialog1.FileName)
   PlayAudio.Play()
End If
```

Ajoutez un deuxième bouton pour la vidéo et le code suivant :

```
If OpenFileDialog1.ShowDialog = Windows.Forms.DialogResult
✂ .OK Then
Dim PlayVideo As New Microsoft.DirectX.AudioVideoPlayback
✂ .Video(OpenFileDialog1.FileName)
PlayVideo.Play()
End If
```

La RichTextBox

Sauvegarder un fichier

```
SaveFileDialog1.Filter = "Fichiers RTF(*.rtf)|.rtf"
If (SaveFileDialog1.ShowDialog() = Windows.Forms
✂ .DialogResult.OK)_
 Then
RichTextBox1.SaveFile(SaveFileDialog1.FileName)
End If
```

Copier du texte dans le presse papier

```
RichTextBox1.Copy()
```

Coller du texte venant du presse papier

```
RichTextBox1.Paste()
```

Couper du texte vers le presse papier

```
RichTextBox1.Cut()
```

Vider le contenu de la zone de texte

```
RichTextBox1.Clear()
```

Charger un fichier

```
RichTextBox1.LoadFile(OpenFileDialog1.FileName)
```

Chapitre 15

Index

G

H

I

J

L

M

N

O

P

Composé en France par Jouve
11, bd de Sébastopol - 75001 Paris